The iOS Apprentice 2

Checklists (Part 2)

By Matthijs Hollemans

Version 1.4

Adding multiple checklists

The app is named Checklist**s** for a reason: it allows you to keep more than one list of to-do items. So far the app has only supported a single list but now we'll give it the capability to handle multiple checklists.

The steps for this section are:

- Add a new screen that shows all the checklists
- Create a screen that lets you add/edit checklists
- Show the to-do items that belong to a particular checklist when you tap the name of that list
- Save all the checklists to a file and load them in again

Two new screens means we'll add two new view controllers: `AllListsViewController` that shows all your lists and `ListDetailViewController` that allows you to add a new list or edit the name and icon of an existing list.

The app's main screen is currently the `ChecklistsViewController`. This file was created by Xcode as part of the Single View Application template and was named after the Class Prefix we chose. In light of the changes we're about to make, the plural form "Checklists" no longer makes sense as this screen only shows the to-do items that belong to a single checklist.

» Use Xcode's Refactor tool to rename this object to `ChecklistViewController` (drop the "s" after "Checklist"). Don't forget to change the Storyboard as well!

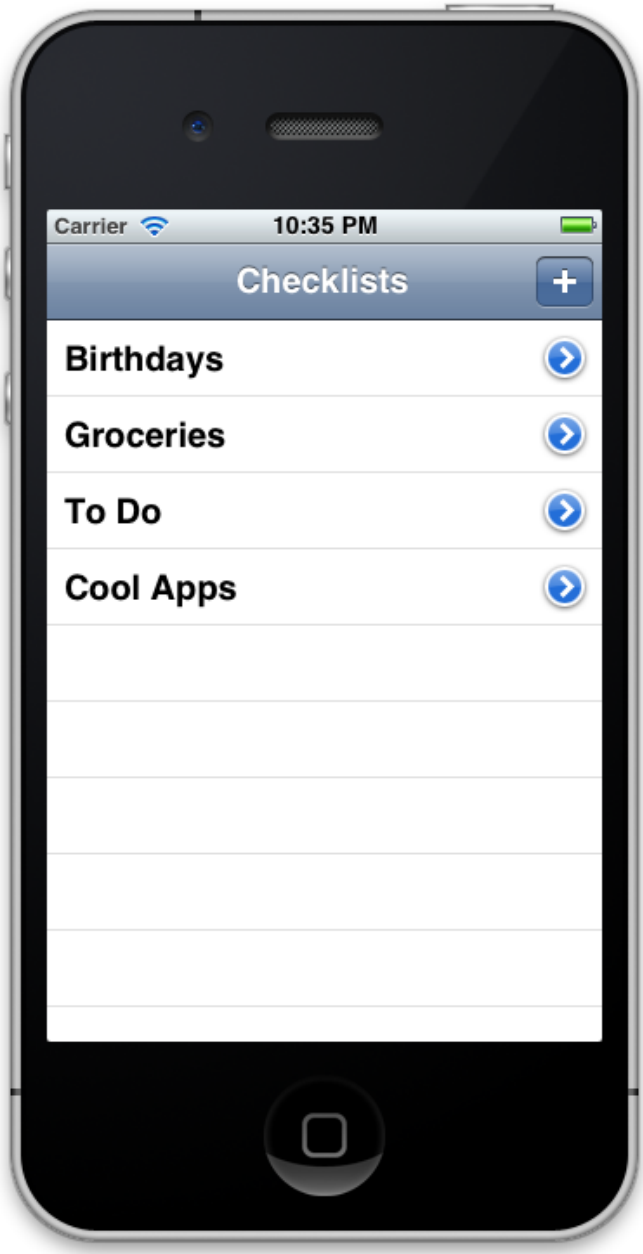
When you're done, do a clean build and run the app to make sure it all still works.

The All Lists screen

We will first add the new `AllListsViewController`. This will now become the main screen of our app.

When we're done this is what it will look like:

The new main screen of our app



This screen is very similar to what we created before. It's a table view controller that shows a list of `Checklist` objects (not `ChecklistItem` objects). From now on, I will refer to this screen as the "All Lists" screen and to the screen that shows the to-do items from a single checklist as the "Checklist" screen.

» Right-click the Checklists group in the Project Navigator and choose New File. Under Cocoa Touch choose the "UIViewController subclass" template (or just "Objective-C class" if you have Xcode 4.3). Press Next and name the new file "AllListsViewCon-

troller". Choose "Subclass of UITableViewController". Uncheck "Targeted for iPad" and also "With XIB for user interface".

The default template for this file needs some work before we can run the app. As a first step, we'll put some fake data in the table view just to get it up and running. I always like to take as small a step as possible, then run the app to see if it's working. Once everything works, we can expand on what we have and put in the real data.

» In AllListsViewController.m, remove the `numberOfSectionsInTableView` method.

» Change the `numberOfRowsInSection` method to:

AllListsViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
{
  return 3;
}
```

» Change `cellForRowAtIndexPath` to:

AllListsViewController.m

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"Cell";

  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:↵
      CellIdentifier];
  if (cell == nil) {
    cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault ↵
        reuseIdentifier:CellIdentifier];
  }

  cell.textLabel.text = [NSString stringWithFormat:@"List %d", indexPath.row];
  return cell;
}
```

This is very similar to what the template already put in that method, except that you're adding this line to put some text in the cells:

```
  cell.textLabel.text = [NSString stringWithFormat:@"List %d", indexPath.row];
```

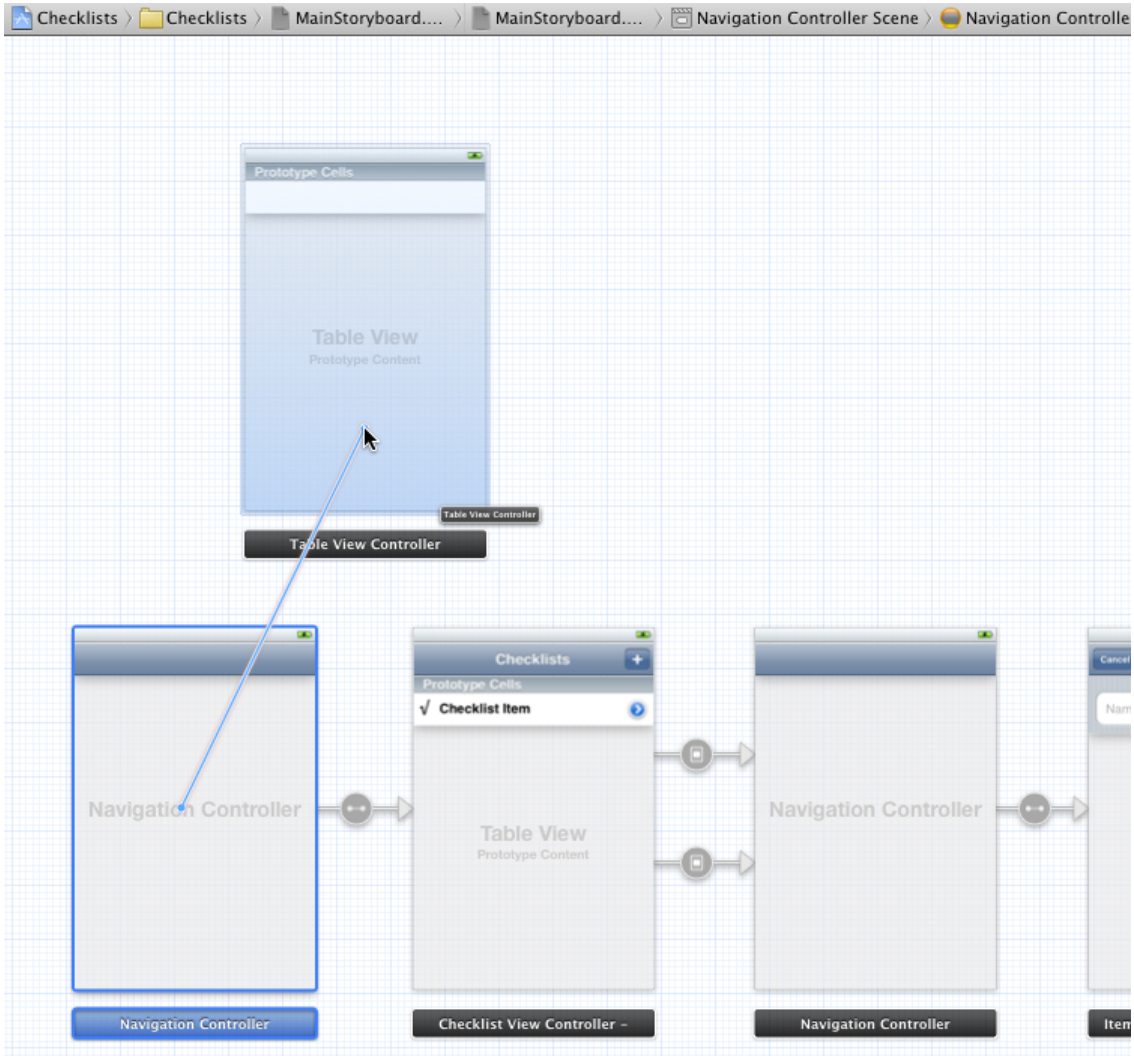
The final step is to add this new view controller to the Storyboard.

» Open the Storyboard editor and drag a new Table View Controller onto the canvas.

» Ctrl-drag from the very first navigation controller to this new table view controller. From the popup menu choose “Relationship - Root View Controller”.

This will break the connection that existed between the navigation controller and the Checklist View Controller so that “Checklists” is no longer the app’s main screen.

Ctrl-drag from the navigation controller to the new table view controller



» Select the new table view controller and set its Class in the Identity Inspector to “AllListsViewController”.

» Double-click the view controller’s navigation bar and change its title to “Checklists”.

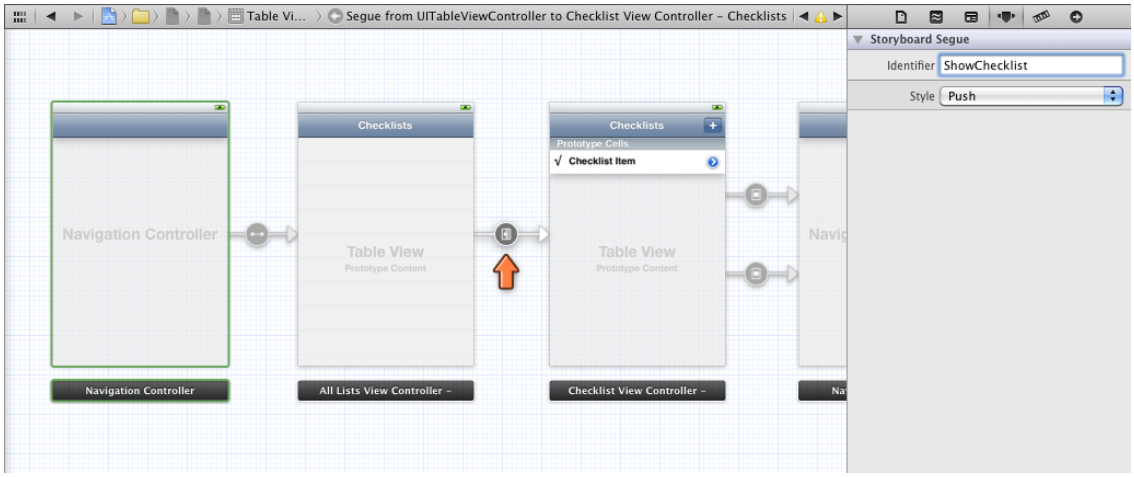
Just for the fun of it, we're not going to use prototype cells for this table view. It would be perfectly fine if you did, and as an exercise you could rewrite the code to use prototype cells later, but I want to show you another way of making table view cells.

» Delete the empty prototype cell from the All Lists View Controller.

» Ctrl-drag from the All Lists View Controller icon in the dock (or the scene list) into the Checklist View Controller and create a segue.

This adds a push transition from the All Lists screen to the Checklist screen. This segue isn't connected to any button or table view cell so we'll have to invoke it manually.

The All Lists View Controller is connected to the Checklist View Controller with a push segue



» Click on the segue to select it, go to the Attributes Inspector and give it the identifier "ShowChecklist". The Style should be Push because we're pushing the Checklist View Controller onto the navigation stack when we perform this segue.

» In AllListsViewController.m, change `didSelectRowAtIndexPath` to the following:

AllListsViewController.m

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
  [self performSegueWithIdentifier:@"ShowChecklist" sender:nil];
}
```

Recall that this table view delegate method is invoked when you tap a row. Previously, a tap on a row would automatically perform the segue because we hooked up the segue to the prototype cell. However, the table view for this screen isn't using prototype cells

and therefore we have to perform the segue manually. That's simple enough: just call `performSegueWithIdentifier` with the name of the segue and things will start moving.

» Run the app. It now looks like this:

Our first version of the All Lists screen (left). Tapping a row opens the Checklist screen (right).

If you tap on a row, the familiar `ChecklistViewController` slides into the screen. You can tap the Checklists button in the top-left to go back to the main list. Now we're truly using the power of the navigation controller!

We're going to duplicate most of the functionality from the `ChecklistViewController` for this new All Lists screen. We'll add a + button at the top that lets you add a new checklist, we'll do swipe-to-delete, and we'll let you edit the name of the checklist from a disclosure button. We'll also save the array of `Checklist` objects to the Checklists.plist

file. Because you've already seen how this works, we'll go through the steps a bit quicker this time.

We begin by creating a data model object that represents a checklist.

» Add a new file to the project based on the "Objective-C class" template, "Subclass of NSObject", and name it "Checklist".

This adds the files Checklist.h and Checklist.m to the project.

» Give Checklist.h a `name` property:

Checklist.h

```
@interface Checklist : NSObject

@property (nonatomic, copy) NSString *name;

@end
```

» Synthesize this property in Checklist.m, below the `@implementation` line:

Checklist.m

```
@synthesize name;
```

That takes care of the `Checklist` object. Next, we'll give `AllListsViewController` an array that will store these new `Checklist` objects.

» Add to AllListsViewController.m, somewhere above the `@implementation` line:

AllListsViewController.m

```
#import "Checklist.h"
```

» Add a new instance variable block with an array named `lists` that will hold the `Checklist` objects:

AllListsViewController.m

```
@implementation AllListsViewController {
  NSMutableArray *lists;
}
```

We have to fill this list up with some test data, which we'll do from the `initWithCoder` method. Remember that this method is automatically invoked by UIKit as it loads the view controller from the Storyboard file.

» Add the `initWithCoder` method:

AllListsViewController.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {

    lists = [[NSMutableArray alloc] initWithCapacity:20];

    Checklist *list;

    list = [[Checklist alloc] init];
    list.name = @"Birthdays";
    [lists addObject:list];

    list = [[Checklist alloc] init];
    list.name = @"Groceries";
    [lists addObject:list];

    list = [[Checklist alloc] init];
    list.name = @"Cool Apps";
    [lists addObject:list];

    list = [[Checklist alloc] init];
    list.name = @"To Do";
    [lists addObject:list];
  }
  return self;
}
```

Notice that the file already has an `initWithStyle` method. You can remove that method as we won't be using it. Since the view controller is part of a Storyboard, it will always be initialized using `initWithCoder`.

» Change the `numberOfRowsInSection` method to return the number of objects in our new array:

AllListsViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
{
  return [lists count];
```

```
}
```

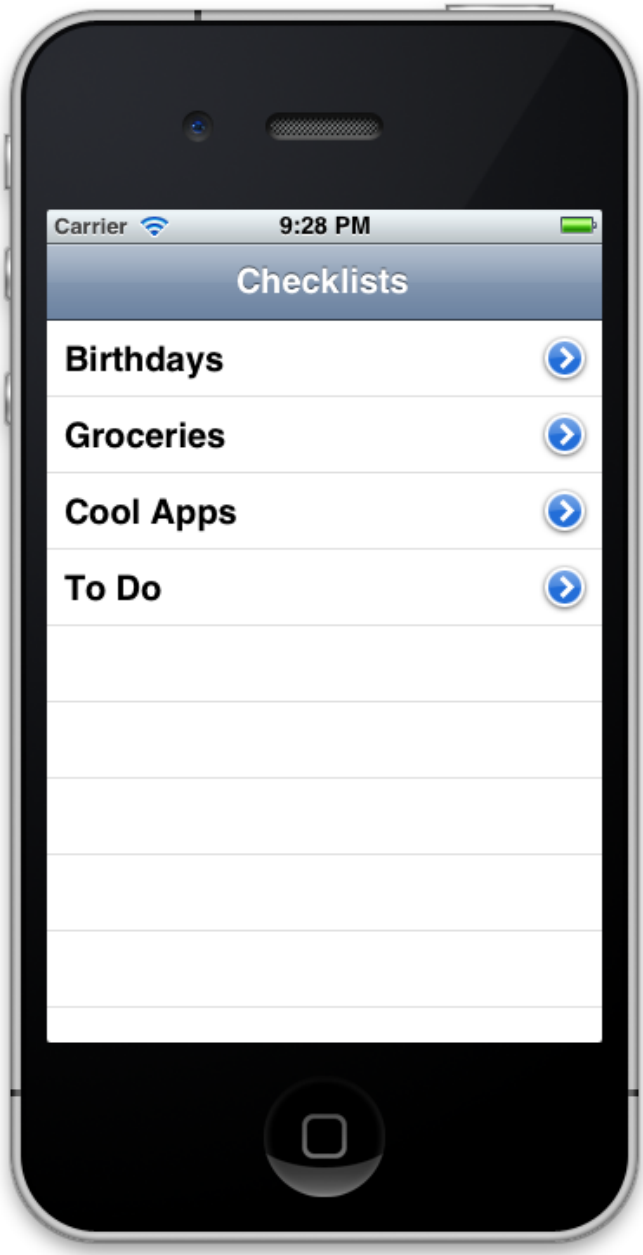
» Finally, change `cellForRowAtIndexPath` to create the cells for the rows:

AllListsViewController.m

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"Cell";

  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:↵
      CellIdentifier];
  if (cell == nil) {
    cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault ↵
        reuseIdentifier:CellIdentifier];
  }

  Checklist *checklist = [lists objectAtIndex:indexPath.row];

  cell.textLabel.text = checklist.name;
  cell.accessoryType = UITableViewCellAccessoryDetailDisclosureButton;
  return cell;
}
```

» Run the app. It looks like this:

The table view shows Checklist objects



The data model now consists of the `lists` array from `AllListsViewController` and the `items` array from `ChecklistViewController`, and the `Checklist` and `ChecklistItem` objects that they respectively contain.

Ways to make table view cells

Our `cellForRowAtIndexPath` method does a lot more than the one from `ChecklistViewController`. There we just did the following to obtain a new table view cell:

```
UITableViewCell *cell =
  [tableView dequeueReusableCellWithIdentifier:@"ChecklistItem"];
```

But here we have a whole chunk of code to accomplish the same:

```
static NSString *CellIdentifier = @"Cell";

UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:↵
    CellIdentifier];
if (cell == nil) {
  cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
                              reuseIdentifier:CellIdentifier];
}
```

The call to `dequeueReusableCellWithIdentifier` is still there, except that previously the Storyboard had a prototype cell with that identifier and now it doesn't. If the table view cannot find a cell to re-use (and it won't until it has enough cells to fill the entire visible area), this method will return `nil` and we have to create our own cell by hand.

There are four ways that you can make table view cells:

1. Using prototype cells. This is a new feature in iOS 5 and very handy. We did this in `ChecklistViewController`.

2. Using static cells. We did this for the Add/Edit Item screen. Also new with iOS 5 and also very handy, but limited to screens where you know in advance which cells you'll have. The big advantage with static cells is that you don't need to provide any of the data source methods (`cellForRowAtIndexPath` and so on).

3. By hand, what we did above. This is how you were supposed to do it before iOS 5 and chances are you'll run across code examples that do it this way, especially from older articles and books. It's a bit more work but also offers you most of the flexibility.

4. Using a nib. The nibs we have seen so far contained an entire view controller but it is also possible to make a nib with just a custom `UITableViewCell` object. This is very similar to using prototype cells, except that you can do it outside of a Storyboard.

When you create a cell by hand you specify a certain *cell style*, which gives you a cell with a preconfigured layout that already has labels and an image view. For the All Lists View Controller we're using the "Default" style but later in this tutorial we'll switch to "Subtitle", which gives us a second, smaller label below the main label.

Using standard cell styles means you don't have to design your own cell layout. For many apps these standard layouts are sufficient so that saves you some work. Prototype cells and static cells can also use these standard cell styles. The default style for a prototype or static cell is "Custom", which requires you to use your own labels, but you can change that to one of the built-in styles in the Storyboard editor.

And finally, a warning: Sometimes I see people writing code that creates a new cell for every row rather than trying to reuse cells. Don't do that! Always ask the table view first whether it has a cell available that can be recycled using `dequeueReusableCellWithIdentifier`. Creating a new cell for each row will cause your app to slow down, as object creation is slower than simply re-using an existing object. Creating all these new objects also takes up more memory, which is a precious commodity on mobile devices. For the best performance, reuse those cells!

You may have noticed that when you tap the name of a checklist, the Checklist screen slides into view but it currently always shows the same to-do items, regardless of which row you tap on. Each checklist should really have its own list of to-do items. We'll work on that later in this tutorial as this requires a significant change to our data model.

As a start, let's set the title of the screen to reflect the chosen checklist.

» Change ChecklistViewController.h to the following:

ChecklistViewController.h

```
#import <UIKit/UIKit.h>
#import "ItemDetailViewController.h"

@class Checklist;

@interface ChecklistViewController : UITableViewController <↵
    ItemDetailViewControllerDelegate>

@property (nonatomic, strong) Checklist *checklist;

@end
```

We've just added a property for a `Checklist` object named `checklist`. Don't forget the forward declaration `@class Checklist;` at the top or Xcode will complain that `ChecklistViewController` doesn't know anything about the `Checklist` object.

» At the top of ChecklistViewController.m, add an import statement to load the complete definition of the `Checklist` object:

ChecklistViewController.m

```
#import "Checklist.h"
```

» Also add a `@synthesize` below the `@implementation` bit:

ChecklistViewController.m

```
@implementation ChecklistViewController {
  NSMutableArray *items;
}

@synthesize checklist;
```

» Change `viewDidLoad` to:

ChecklistViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];
  self.title = self.checklist.name;
}
```

This changes the title of the screen (which is shown in the navigation bar) to the name of the `Checklist` object.

Now all we have to do is give this `Checklist` object to the `ChecklistViewController` when the segue is performed.

» In `AllListsViewController`, update `didSelectRowAtIndexPath` to the following:

AllListsViewController.m

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
  Checklist *checklist = [lists objectAtIndex:indexPath.row];

  [self performSegueWithIdentifier:@"ShowChecklist" sender:checklist];
}
```

As before, we use `performSegueWithIdentifier` to start the segue. This method has a `sender` parameter that we previously set to `nil`. Now we'll use it to send along the `Checklist` object from the row that the user tapped on.

You can put anything you want into `sender`. If the segue is performed by the Storyboard (rather than manually like we do here) then `sender` will refer to the control that triggered it, for example the `UIBarButtonItem` for the Add button or the `UITableViewCell` for a row in the table. But because we start this particular segue by ourselves, we can put into `sender` whatever is most convenient for us.

Putting the `Checklist` object into the `sender` parameter doesn't give this object to the `ChecklistViewController` yet. That happens in `prepareForSegue`.

» Add the following method below `didSelectRowAtIndexPath`:

AllListsViewController.m

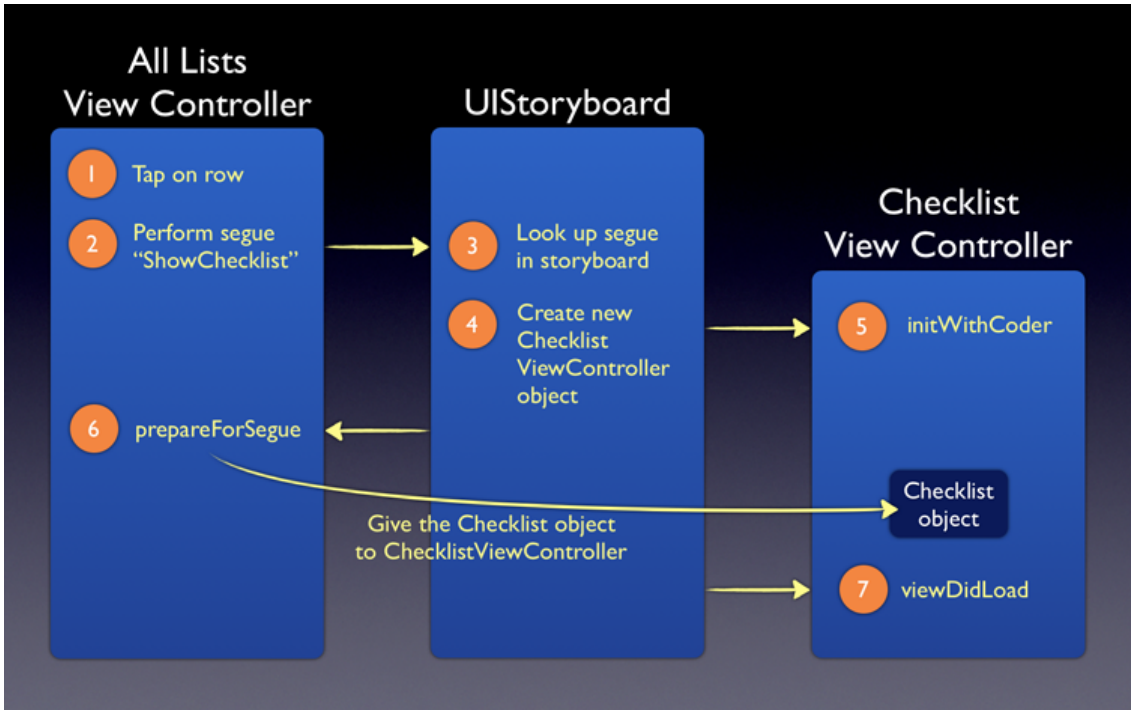
```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"ShowChecklist"]) {
    ChecklistViewController *controller = segue.destinationViewController;
    controller.checklist = sender;
  }
}
```

We've seen this method already. `prepareForSegue` is called by the Storyboard right before the segue happens. Here we get a chance to set the properties of the new view controller before it will become visible.

We need to give it the `Checklist` object from the row that the user tapped. That's why we put that object in the `sender` parameter earlier. We could have temporarily stored the `Checklist` object in an ivar instead but passing it along in the `sender` parameter is much easier.

All of this happens before `ChecklistViewController`'s view is loaded, so `viewDidLoad` can set the title of the screen accordingly.

The sequence of events involved in performing a segue



» Add an import at the top of AllListsViewController.m, so the compiler can find the `ChecklistViewController` object:

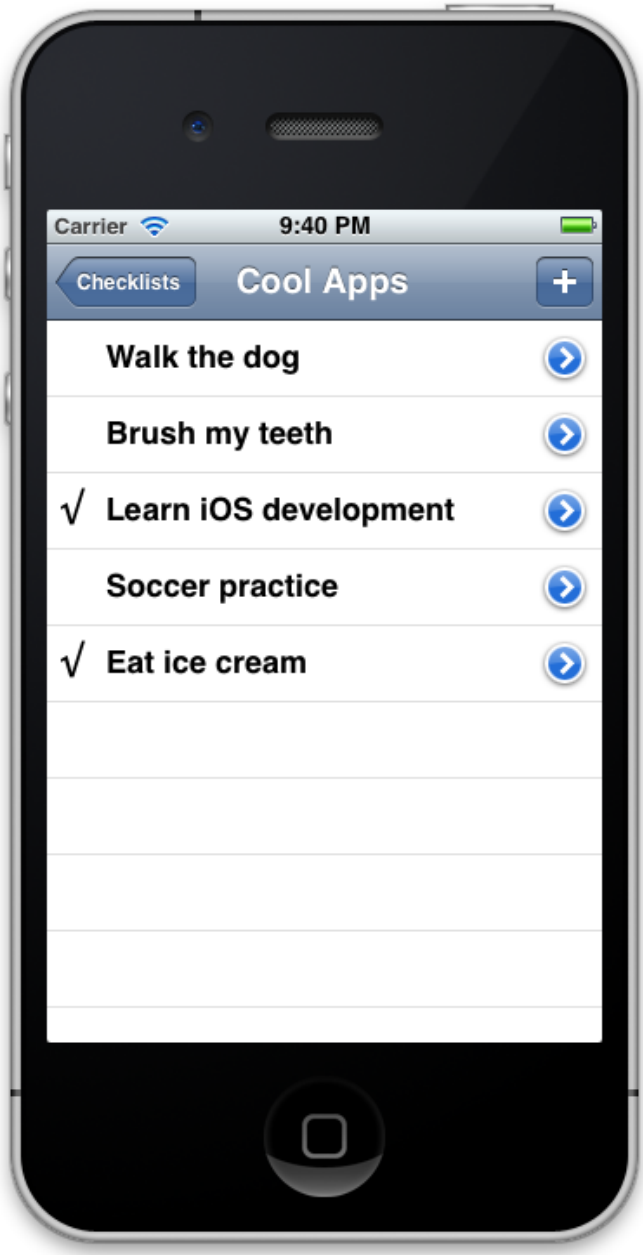
AllListsViewController.m

```
#import "ChecklistViewController.h"
```

That should do it.

» Run the app and notice that when you tap the row for a checklist, the next screen properly takes over the title.

The name of the chosen checklist now appears in the navigation bar



Note that giving the `Checklist` object to the `ChecklistViewController` does not make a copy of it. We only pass the view controller a reference (also known as a *pointer*) to that object, so any changes we make to it are also seen by `AllListsViewController`. Both view controllers have access to the exact same `Checklist` object. We'll use that to our advantage later in order to add new `ChecklistItems` to the `Checklist`.

Adding and editing checklists

Let's quickly add the Add Checklist / Edit Checklist screen. This is going to be yet another `UITableViewController`, this time with static cells, and we'll present it modally from the `AllListsViewController`. If the previous sentence made perfect sense to you, then you're getting the hang of this!

» Add a new file to the project, a `UITableViewController` subclass named "ListDetailViewController".

» Change ListDetailViewController.h to:

ListDetailViewController.h

```
#import <UIKit/UIKit.h>

@class ListDetailViewController;
@class Checklist;

@protocol ListDetailViewControllerDelegate <NSObject>
- (void)listDetailViewControllerDidCancel:(ListDetailViewController *)↵
    controller;
- (void)listDetailViewController:(ListDetailViewController *)controller ↵
    didFinishAddingChecklist:(Checklist *)checklist;
- (void)listDetailViewController:(ListDetailViewController *)controller ↵
    didFinishEditingChecklist:(Checklist *)checklist;
@end

@interface ListDetailViewController : UITableViewController <↵
    UITextFieldDelegate>

@property (nonatomic, strong) IBOutlet UITextField *textField;
@property (nonatomic, strong) IBOutlet UIBarButtonItem *doneBarButton;
@property (nonatomic, weak) id <ListDetailViewControllerDelegate> delegate;
@property (nonatomic, strong) Checklist *checklistToEdit;

- (IBAction)cancel;
- (IBAction)done;

@end
```

This may seem like a lot of code all at once, but I simply took the contents of ItemDetailViewController.h and changed the names. Also, instead of a `ChecklistItem` we're now dealing with a `Checklist`.

» To the top of ListDetailViewController.m add:

ListDetailViewController.m

```
#import "Checklist.h"
```

» Below the `@implementation` line add:

ListDetailViewController.m

```
@synthesize textField;
@synthesize doneBarButton;
@synthesize delegate;
@synthesize checklistToEdit;
```

» Change `viewDidLoad` to:

ListDetailViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  if (self.checklistToEdit != nil) {
    self.title = @"Edit Checklist";
    self.textField.text = self.checklistToEdit.name;
    self.doneBarButton.enabled = YES;
  }
}
```

» And change (or add if it doesn't exist yet) the `viewWillAppear` method:

ListDetailViewController.m

```
- (void)viewWillAppear:(BOOL)animated
{
  [super viewWillAppear:animated];

  [self.textField becomeFirstResponder];
}
```

» Throw away everything below the `#pragma mark Table view data source` line and replace it with:

ListDetailViewController.m

```
- (IBAction)cancel
{
```

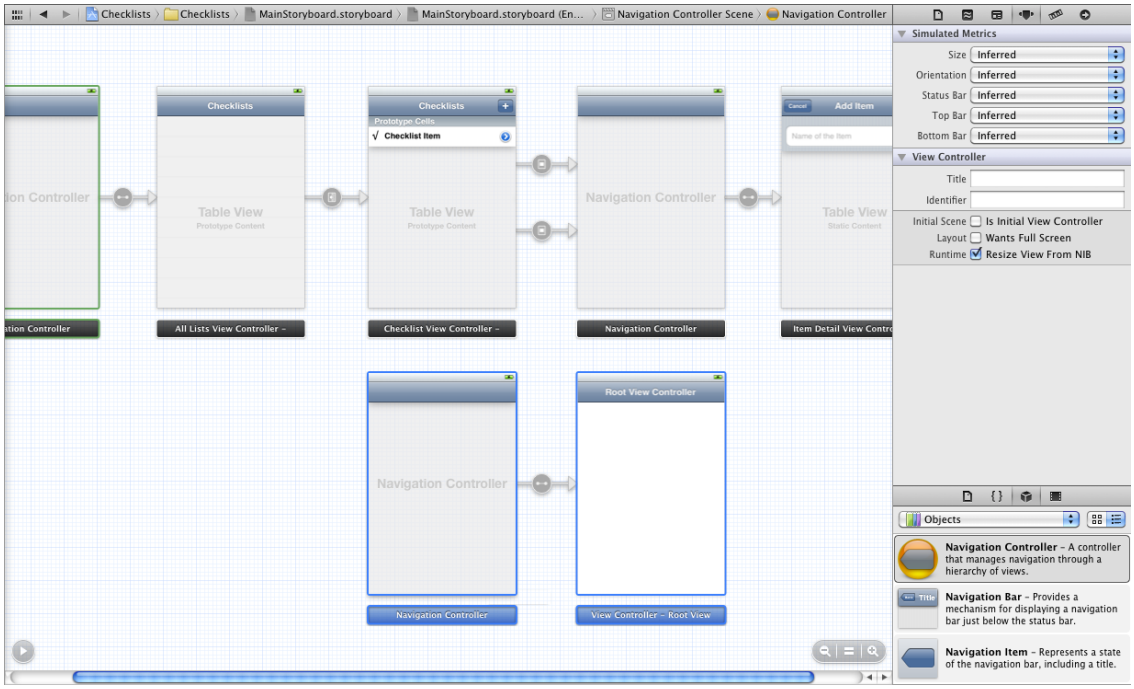
```
  [self.delegate listDetailViewControllerDidCancel:self];
}

- (IBAction)done
{
  if (self.checklistToEdit == nil) {
    Checklist *checklist = [[Checklist alloc] init];
    checklist.name = self.textField.text;

    [self.delegate listDetailViewController:self didFinishAddingChecklist:↵
        checklist];
  } else {
    self.checklistToEdit.name = self.textField.text;
    [self.delegate listDetailViewController:self didFinishEditingChecklist:self↵
        .checklistToEdit];
  }
}

- (NSIndexPath *)tableView:(UITableView *)tableView willSelectRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  return nil;
}

- (BOOL)textField:(UITextField *)theTextField shouldChangeCharactersInRange:(↵
    NSRange)range replacementString:(NSString *)string
{
  NSString *newText = [theTextField.text stringByReplacingCharactersInRange:↵
      range withString:string];
  self.doneBarButton.enabled = ([newText length] > 0);
  return YES;
}
```

Again, this is what we did in `ItemDetailViewController` but now for `Checklist` objects instead of `ChecklistItem` objects.

Let's make the user interface for this new view controller in the Storyboard editor.

» Go to the Storyboard editor. Drag a new Navigation Controller from the Object Library into the canvas and move it below the other view controllers.

Dragging a new navigation controller into the canvas



» Delete the “Root View Controller” that is attached to the new navigation controller. The Storyboard editor automatically added this second view controller but we don’t need it.

» Select the existing Item Detail View Controller. Press Cmd+D to create a duplicate. Move the duplicate next to the new navigation controller. The navigation bar with the Cancel and Done buttons has disappeared from this duplicate but that’s no problem.

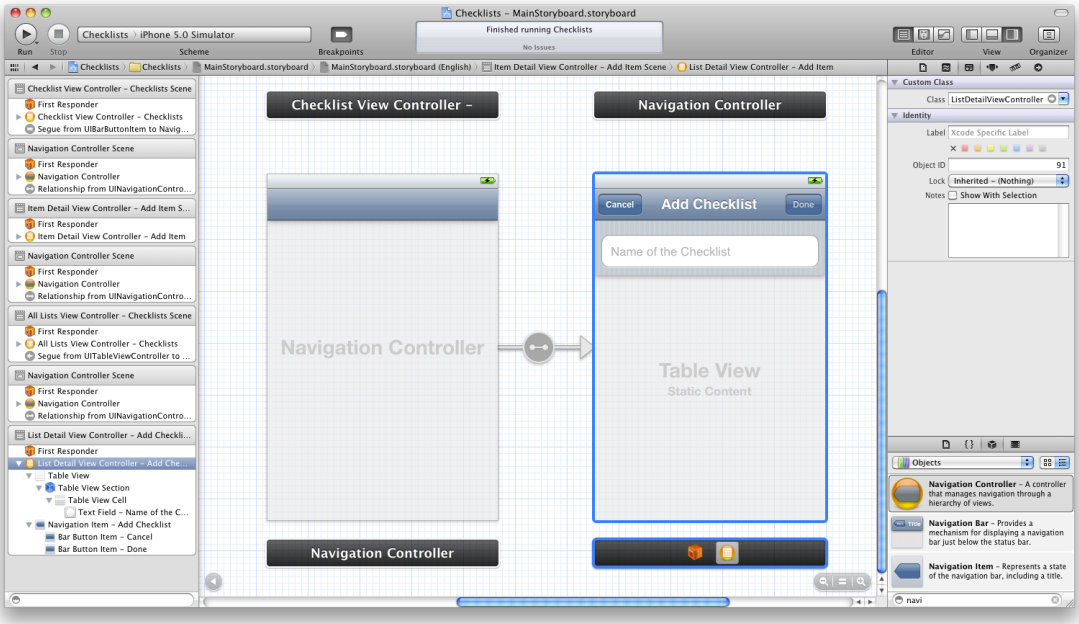
» Ctrl-drag from the navigation controller into this second Item Detail View Controller and choose “Relationship - Root View Controller”. Now it is hooked up again and the navigation bar with the Cancel/Done buttons has reappeared.

» Select the clone of the Item Detail View Controller and go to the Identity Inspector. Change its class to “ListDetailViewController”.

» Change the navigation bar title to “Add Checklist” and the placeholder text in the text field to “Name of the Checklist”.

This completes the steps for converting this view controller to the Add / Edit Checklist screen:

The List Detail View Controller is now hooked up to the new navigation controller



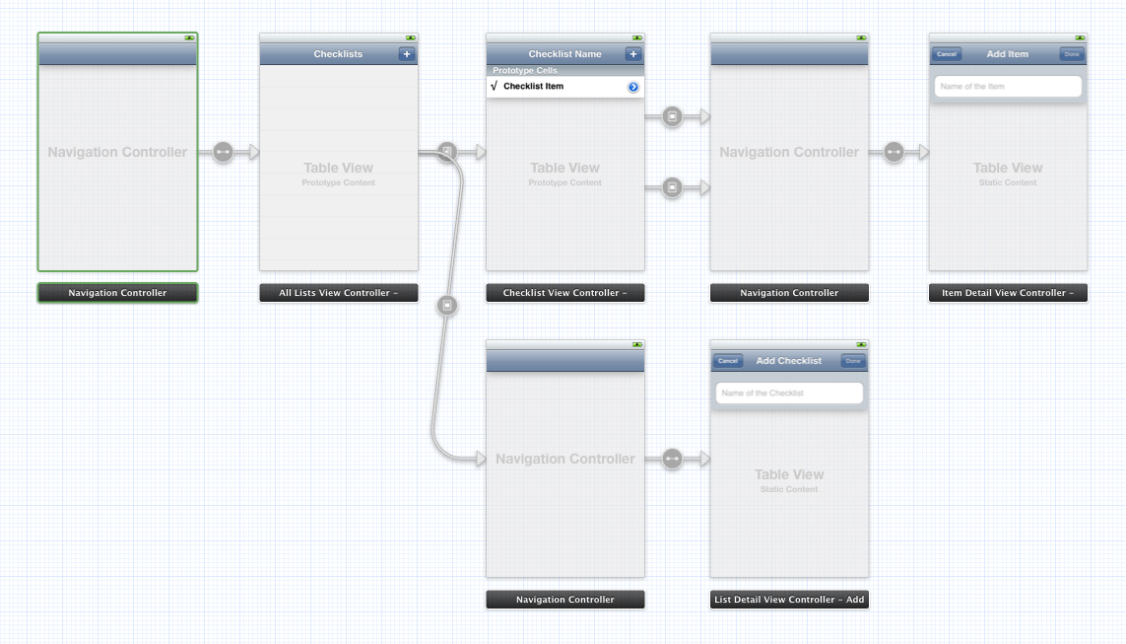
» Go to the All Lists View Controller and drag a Bar Button Item into its navigation bar. Change it into an Add button. Ctrl-drag from this button to the navigation controller below to add a new Modal segue.

» Click on the new segue and name it “AddChecklist”.

» Just so we don’t get confused as to which screen does what, change the title of the Checklist View Controller from “Checklists” to “Checklist Name”.

Our storyboard now looks like this:

The full storyboard



Almost there. We still have to make the `AllListsViewController` the delegate for the `ListDetailViewController` and then we're done. Again, it's very similar to what we did before.

» In AllListsViewController.m, extend `prepareForSegue` to:

AllListsViewController.m

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"ShowChecklist"]) {
    ChecklistViewController *controller = segue.destinationViewController;
    controller.checklist = sender;
  } else if ([segue.identifier isEqualToString:@"AddChecklist"]) {
    UINavigationController *navigationController = segue.↵
        destinationViewController;
    ListDetailViewController *controller = (ListDetailViewController *)↵
        navigationController.topViewController;
    controller.delegate = self;
    controller.checklistToEdit = nil;
  }
}
```

The first `if` doesn't change. We've added a second `if` for the "AddChecklist" segue that we just defined in the Storyboard editor. As before, we look for the view controller inside the navigation controller (which is the `ListDetailViewController`) and set its `delegate` property to `self`.

» At the bottom of the AllListsViewController.m, implement the following delegate methods. You'll get some error messages from Xcode because it doesn't know anything about `ListDetailViewController` here. We'll fix this in a minute with an `#import` so just ignore the errors for now.

AllListsViewController.m

```
- (void)listDetailViewControllerDidCancel:(ListDetailViewController *)↵
    controller
{
  [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)listDetailViewController:(ListDetailViewController *)controller ↵
    didFinishAddingChecklist:(Checklist *)checklist
{
  int newRowIndex = [lists count];
  [lists addObject:checklist];

  NSIndexPath *indexPath = [NSIndexPath indexPathForRow:newRowIndex inSection↵
      :0];
  NSArray *indexPaths = [NSArray arrayWithObject:indexPath];
  [self.tableView insertRowsAtIndexPaths:indexPaths withRowAnimation:↵
      UITableViewRowAnimationAutomatic];

  [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)listDetailViewController:(ListDetailViewController *)controller ↵
    didFinishEditingChecklist:(Checklist *)checklist
{
  int index = [lists indexOfObject:checklist];
  NSIndexPath *indexPath = [NSIndexPath indexPathForRow:index inSection:0];
  UITableViewCell *cell = [self.tableView cellForRowAtIndexPath:indexPath];
  cell.textLabel.text = checklist.name;

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

None of this code should surprise you. It's exactly what we did before but now for the `ListDetailViewController` and `Checklist` objects. These methods are called when the user presses Cancel or Done inside the new Add/Edit Checklist screen.

» Also add the table view data source method that allows the user to delete checklists:

AllListsViewController.m

```
- (void)tableView:(UITableView *)tableView commitEditingStyle:(↵
    UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)↵
    indexPath
```

```
{
  [lists removeObjectAtIndex:indexPath.row];

  NSArray *indexPaths = [NSArray arrayWithObject:indexPath];
  [tableView deleteRowsAtIndexPaths:indexPaths withRowAnimation:↵
      UITableViewRowAnimationAutomatic];
}
```

» Finally, declare this view controller to conform to the delegate protocol by adding `<ListDetailViewControllerDelegate>` to its `@interface` line. This also takes care of the Xcode errors.

AllListsViewController.h

```
#import <UIKit/UIKit.h>
#import "ListDetailViewController.h"

@interface AllListsViewController : UITableViewController <↵
    ListDetailViewControllerDelegate>

@end
```

» Run the app. Now you can add new checklists and delete them again, but you can't edit the names of existing lists yet. That requires one last addition to the code.

To bring up the Edit Checklist screen, the user taps the blue accessory button. In the `ChecklistViewController` we did that by triggering a segue that was defined on the view controller itself (rather than on a specific control). We could do that here too, but I want to show you another way. This time we're not going to use a segue at all, but load the new view controller by hand from the Storyboard.

» Add the accessory-tapped delegate method to AllListsViewController.m:

AllListsViewController.m

```
- (void)tableView:(UITableView *)tableView ↵
    accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
{
  UINavigationController *navigationController = [self.storyboard ↵
      instantiateViewControllerWithIdentifier:@"ListNavigationController"];

  ListDetailViewController *controller = (ListDetailViewController *)↵
      navigationController.topViewController;
  controller.delegate = self;

  Checklist *checklist = [lists objectAtIndex:indexPath.row];
```

```
  controller.checklistToEdit = checklist;

  [self presentViewController:navigationController animated:YES completion:nil↵
      ];
}
```

Inside this method we create the view controller object for the Add/Edit Checklist screen and show it ("present" it) on the screen. In the Bull's Eye app we did something similar with the About screen, except there we loaded the view controller directly from a nib. This time the view controller is embedded in a Storyboard, so we have to ask the Storyboard object to load it for us.

Where did we get that Storyboard object from? As it happens, each view controller has a `self.storyboard` property that refers to the Storyboard the view controller was loaded from. We can use that property to do other things with the Storyboard as well, such as instantiating other view controllers.

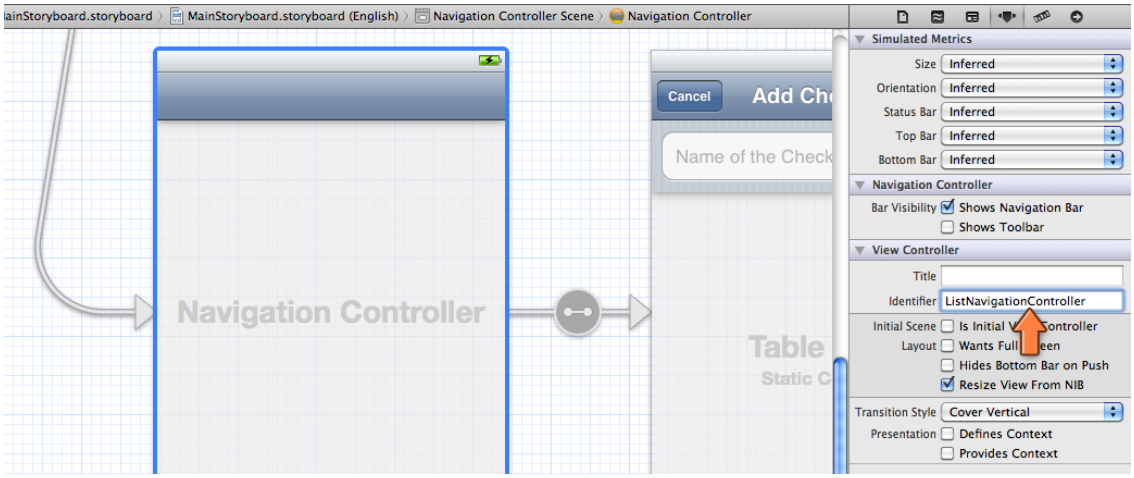
The call to `instantiateViewControllerWithIdentifier` takes an identifier string, `@"ListNavigationController"`. That is how we ask the storyboard to create the new view controller. In our case, this will be the navigation controller that contains the `ListDetailViewController`. We could instantiate the `ListDetailViewController` directly, but we designed it to work inside the navigation controller so that wouldn't make much sense (it would no longer have a title bar or Cancel and Done buttons).

We still have to set this identifier on the navigation controller, otherwise the Storyboard cannot find it.

» Open the Storyboard editor and select the navigation controller that points to List Detail View Controller.

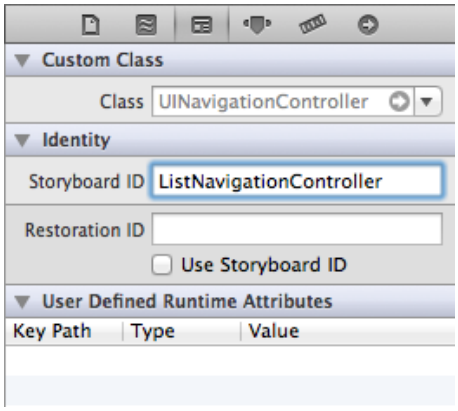
If you're still using Xcode version 4.2, 4.3 or 4.4, go to the Attributes Inspector and type "ListNavigationController" into the Identifier field:

Setting an identifier on the navigation controller in Xcode 4.4 or lower



On Xcode 4.5 this field has moved into the Identity inspector and is called Storyboard ID:

Setting an identifier on the navigation controller in Xcode 4.5



» That should do the trick. Now run the app and tap some detail disclosure buttons.

Exercise: Set the "ListNavigationController" identifier on the List Detail View Controller instead and see what happens when you run the app. ■

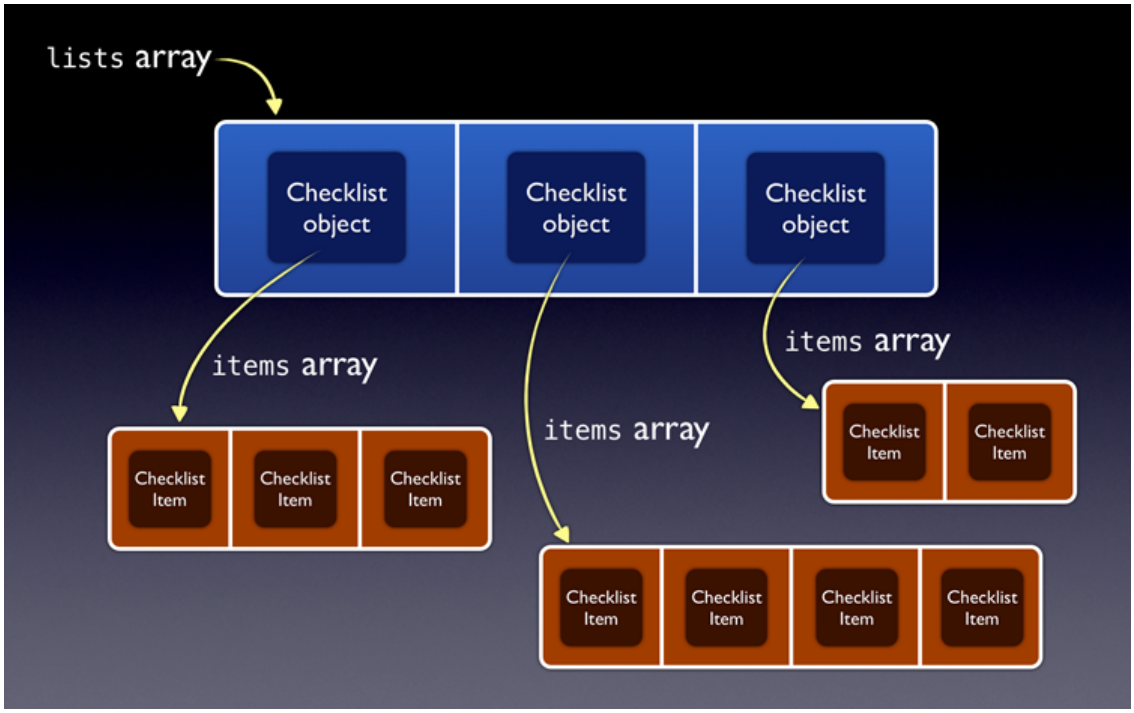
You can find the project files for the app up to this point under "07 - Lists" in the tutorial's Source Code folder.

Putting items into checklists

This is all well and good, but checklists don't actually contain any to-do items yet. So far the list of to-do items and the actual checklists have been separate from each other.

Let's change our data model to look like this:

Each Checklist object has an array of ChecklistItem objects



There will still be a `lists` array that contains the `Checklist` objects, but each of these `Checklists` will have its own array of `ChecklistItem` objects.

» Add a new property to Checklist.h:

Checklist.h

```
@property (nonatomic, strong) NSMutableArray *items;
```

» Synthesize it in Checklist.m:

Checklist.m

```
@synthesize items;
```

» Change `Checklist`'s `init` method to the following:

Checklist.m

```
- (id)init
{
  if ((self = [super init])) {
    self.items = [[NSMutableArray alloc] initWithCapacity:20];
  }
  return self;
}
```

The `Checklist` object will now contain the array of `ChecklistItem` objects. Initially, that array will be empty.

Earlier we fixed `prepareForSegue` so that when you tap on a row in the main screen, we segue into the `ChecklistViewController` and the `Checklist` object that belongs to that row is passed along. Currently `ChecklistViewController` still gets the `ChecklistItem` objects from its own `items` array but we will change that so it reads from the `items` array inside that `Checklist` object instead.

» Make the following changes in ChecklistViewController.m:

ChecklistViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
{
  return [self.checklist.items count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  . . .

  ChecklistItem *item = [self.checklist.items objectAtIndex:indexPath.row];

  . . .
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
     *)indexPath
{
  . . .

  ChecklistItem *item = [self.checklist.items objectAtIndex:indexPath.row];

  . . .
}
```

```
- (void)tableView:(UITableView *)tableView commitEditingStyle:(↵
    UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)↵
    indexPath
{
  [self.checklist.items removeObjectAtIndex:indexPath.row];

  . . .
}

- (void)tableView:(UITableView *)tableView ↵
    accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
{
  . . .

  ChecklistItem *item = [self.checklist.items objectAtIndex:indexPath.row];

  . . .
}

- (void)itemDetailViewController:(ItemDetailViewController *)controller ↵
    didFinishAddingItem:(ChecklistItem *)item
{
  int newRowIndex = [self.checklist.items count];
  [self.checklist.items addObject:item];

  . . .
}

- (void)itemDetailViewController:(ItemDetailViewController *)controller ↵
    didFinishEditingItem:(ChecklistItem *)item
{
  int index = [self.checklist.items indexOfObject:item];

  . . .
}
```

Anywhere it said `items` you have changed it to say `self.checklist.items` instead.

» Delete the following methods from ChecklistViewController.m:

- `(NSString *)documentsDirectory`
- `(NSString *)dataFilePath`
- `(void)saveChecklistItems`
- `(void)loadChecklistItems`
- `(id)initWithCoder:(NSCoder *)aDecoder`

We recently added these methods to load and save the checklist items from a file. That is no longer the responsibility of this view controller, though. It is better for our design if we make the `Checklist` object do that. Loading and saving data model objects really belongs in the data model itself, rather than in a controller.

But before we get to that, let's first test whether our changes were successful. Xcode is complaining about 4 errors because we still call the method `saveChecklistItems` at several places in our code. We should remove those lines as we will soon be saving the items in a different place.

» Remove the lines that call `saveChecklistItems`.

» Also remove the `items` ivar from the `@implementation`'s variable section at the top.

This:

```
@implementation ChecklistViewController {
  NSMutableArray *items;
}
```

simply becomes again:

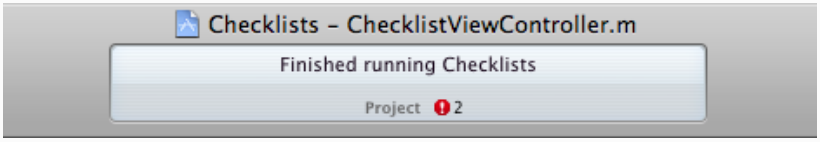
```
@implementation ChecklistViewController
```

Since there are no ivars anymore, the { } brackets are no longer necessary. Note that unlike regular statements, the `@implementation` line never ends in a semicolon!

Xcode errors

When Xcode detects a problem it shows a warning or error icon at the top of the window:

Xcode shows an error icon



When you fix the problem, this error icon may not immediately go away. Xcode is pretty smart about detecting any changes you make, but it doesn't always pick up on everything. At times that may be a bit confusing. After all, you just fixed the problem but Xcode still complains about it.

Just press Run to launch the app or press Cmd+B to do a build without running the app. If there are truly still errors or warnings then Xcode will tell you.

Let's add some fake data into the various `Checklist` objects so that we can test whether this new design actually works. In `AllListsViewController`'s `initWithCoder` method we already put fake `Checklist` objects into the `lists` array. It's time to add something new to this method.

» At the top of the AllListsViewController.m file, add an import:

AllListsViewController.m

```
#import "ChecklistItem.h"
```

» Change the `initWithCoder` method to the following:

AllListsViewController.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    lists = [[NSMutableArray alloc] initWithCapacity:20];

    Checklist *list;

    list = [[Checklist alloc] init];
    list.name = @"Birthdays";
    [lists addObject:list];

    list = [[Checklist alloc] init];
    list.name = @"Groceries";
    [lists addObject:list];
```

```
    list = [[Checklist alloc] init];
    list.name = @"Cool Apps";
    [lists addObject:list];

    list = [[Checklist alloc] init];
    list.name = @"To Do";
    [lists addObject:list];

    for (Checklist *list in lists) {
      ChecklistItem *item = [[ChecklistItem alloc] init];
      item.text = [NSString stringWithFormat:@"Item for %@", list.name];
      [list.items addObject:item];
    }
  }
  return self;
}
```

The only new bit is this:

```
    for (Checklist *list in lists) {
      ChecklistItem *item = [[ChecklistItem alloc] init];
      item.text = [NSString stringWithFormat:@"Item for %@", list.name];
      [list.items addObject:item];
    }
```

This introduces something we haven't seen before: the `for`-statement. Like `if`, this is a special language construct.

Programming language constructs

For the sake of review, let's go over the programming language stuff we've already seen. Most modern programming languages offer at least the following basic building blocks:

- The ability to remember values by storing things into variables. Some variables are simple, such as `int` and `BOOL`. Others can store objects (`UIButton`, `ChecklistItem`) and even others can store collections of objects (`NSMutableArray`).
- The ability to read values from variables and use them for basic arithmetic (multiply, add) and comparisons (greater than, not equals, etc).
- The ability to make decisions. We've already seen the `if`-statement, but there is also a `switch` statement that is shorthand for an `if` with many `else if`s.

- The ability to group functionality into units such as methods and functions. You can call those methods and receive back a result value that you can then use in further computations.
- The ability to group functionality (methods) and data (variables) into objects.
- The ability to repeat a set of statements more than once. This is what the `for` statement does. There are several other ways to perform repetitions as well: `while` and `do while`. Endlessly repeating things is what computers are good at.

Everything else is built on top of these building blocks. We've seen most of these already, but repetitions (or *loops* in programmer slang) are new. If you grok these concepts, then you're well on your way to becoming a software developer.

Let's go through that `for` loop line-by-line:

```
for (Checklist *list in lists) {
  . . .
}
```

This means the following: for every `Checklist` object in the `lists` array, perform the statements that are in between the curly braces.

The first time through the loop the `list` variable will hold a reference to the Birthdays checklist as that is the first `Checklist` object that we created and added to the `lists` array.

Then we do:

```
ChecklistItem *item = [[ChecklistItem alloc] init];
item.text = [NSString stringWithFormat:@"Item for %@", list.name];
[list.items addObject:item];
```

This shouldn't be too unfamiliar. We first create a new `ChecklistItem` object. Then we set its `text` property to "Item for Birthdays" because we replace the `%@` placeholder with the name of the `Checklist` object (`list.name`, which is "Birthdays"). Finally, we add this new `ChecklistItem` to the Birthdays checklist object, or rather, to its `items` array.

That concludes the first pass through this loop. Now the `for`-statement will look at the `lists` array again and sees that there are three more `Checklist` objects in that list. So it puts the next one, Groceries, into the `list` variable and the process repeats. This time the text is "Item for Groceries", which will be put into its own `ChecklistItem` object that goes into the `items` array of the Groceries `Checklist` object.

After that, we add a new `ChecklistItem` with the text "Item for Cool Apps" to the Cool Apps checklist, and an item "Item for To Do" to the To Do checklist. Then there are no more objects left to look at in the `lists` array and the loop ends.

Using loops will often save us a lot of time. We could have written this code as follows:

```
    ChecklistItem *item;

    list = [lists objectAtIndex:0];
    item = [[ChecklistItem alloc] init];
    item.text = @"Item for Birthdays";
    [list.items addObject:item];

    list = [lists objectAtIndex:1];
    item = [[ChecklistItem alloc] init];
    item.text = @"Item for Groceries";
    [list.items addObject:item];

    list = [lists objectAtIndex:2];
    item = [[ChecklistItem alloc] init];
    item.text = @"Item for Cool Apps";
    [list.items addObject:item];

    list = [lists objectAtIndex:3];
    item = [[ChecklistItem alloc] init];
    item.text = @"Item for To Do";
    [list.items addObject:item];
```

That's a little repetitive, which is a good sign it's better to use a loop. And what if we had 100 `Checklist` objects? Would you be willing to copy-paste that code a hundred times? I'd rather use a loop.

Most of the time you won't even know in advance how many objects you'll have, so it's impossible to write it all out by hand. By using a loop you don't need to worry about that. The loop will work just as well for three items as for three hundred. As you can imagine, loops and arrays work quite well together.

» Run the app. You'll see that each checklist now has its own set of items. Play with it for a minute, remove items, add items, and verify that each list indeed is completely separate from the others.

Each Checklist now has its own items

Let's put the load/save code back in. This time we will make `AllListsViewController` do the loading and saving.

» Add the following to AllListsViewController.m, above `initWithCoder`:

AllListsViewController.m

```
- (NSString *)documentsDirectory
{
```

```
  NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, ↵
      NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths objectAtIndex:0];
  return documentsDirectory;
}

- (NSString *)dataFilePath
{
  return [[self documentsDirectory] stringByAppendingPathComponent:@"Checklists↵
      .plist"];
}

- (void)saveChecklists
{
  NSMutableData *data = [[NSMutableData alloc] init];
  NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc] ↵
      initForWritingWithMutableData:data];
  [archiver encodeObject:lists forKey:@"Checklists"];
  [archiver finishEncoding];
  [data writeToFile:[self dataFilePath] atomically:YES];
}

- (void)loadChecklists
{
  NSString *path = [self dataFilePath];
  if ([[NSFileManager defaultManager] fileExistsAtPath:path]) {
    NSData *data = [[NSData alloc] initWithContentsOfFile:path];
    NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc] ↵
        initForReadingWithData:data];
    lists = [unarchiver decodeObjectForKey:@"Checklists"];
    [unarchiver finishDecoding];
  } else {
    lists = [[NSMutableArray alloc] initWithCapacity:20];
  }
}
```

This is mostly identical to what we had before in `ChecklistViewController`, except that we load and save the `lists` array instead of the `items` array.

» Change `initWithCoder` to:

AllListsViewController.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    [self loadChecklists];
  }
  return self;
}
```

This gets rid of the test data we put there earlier and makes the `loadChecklists` method do all the work.

We also have to make the `Checklist` object compliant with `NSCoding`.

» Add the `NSCoding` protocol in Checklist.h:

Checklist.h

```
@interface Checklist : NSObject <NSCoding>
```

» Add the following methods to Checklist.m:

Checklist.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super init])) {
    self.name = [aDecoder decodeObjectForKey:@"Name"];
    self.items = [aDecoder decodeObjectForKey:@"Items"];
  }
  return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
  [aCoder encodeObject:self.name forKey:@"Name"];
  [aCoder encodeObject:self.items forKey:@"Items"];
}
```

Both the `name` and `items` properties are objects (`NSString` and `NSMutableArray`) so we use `decodeObjectForKey:` and `encodeObject:forKey:` to load and save them.

» Before you run the app, remove the old Checklists.plist file from the Simulator's Documents folder. If you don't, the app might crash because the internal format of the file no longer corresponds to the way we're loading and saving.

Weird crashes

When I first wrote this tutorial, I didn't think to remove the file before running the app. That was a mistake but the app actually worked quite fine… until I added a new checklist. At that point the app aborted with the following error message:

```
*** Terminating app due to uncaught exception 'NSRangeException', reason: '***
-[NSMutableIndexSet addIndexesInRange:]: Range {2147483647, 1} exceeds maximum
index value of NSNotFound - 1'
```

The line where the crash occurred was:

```
[self.tableView insertRowsAtIndexPaths:indexPaths
                      withRowAnimation:UITableViewRowAnimationAutomatic];
```

That is a very strange error message and I started to wonder whether I tested the code properly. But then I thought of the old file, removed it and ran the app again. It worked perfectly. Just to make sure it was the fault of that file, I put a copy of the old file back and ran the app again. Sure enough, when I tried to add a new checklist it crashed.

The explanation for this error is that somehow the code manages to load the old file, even though its format is all wrong and no longer corresponds to our data model, but that this puts the table view into a bad state. Any subsequent operations on the table view will cause the app to crash.

You'll run into this type of bug every so often, where the crash isn't directly caused by what you're doing but by something that went wrong earlier on. These kinds of bugs can be tricky to solve, because you can't fix them until you find the true cause.

I'll devote a big section to debugging techniques in a later tutorial because it's inevitable that you'll introduce bugs in your code and knowing how to find and eradicate them is an essential skill that any programmer should master (if only to save you a lot of time and aggravation!).

» Run the app and add a checklist and a few to-do items. Exit the app (with the Stop button) and run it again. You'll see that the list is empty again.

You can add all the checklists and items you want, but nothing gets saved anymore. What's going on here?

Doing saves differently

Previously, we saved our data whenever the user changed something: added a new item, deleted an item, toggled a checkmark. That all happened in `ChecklistViewController`. However, we moved the saving logic into `AllListsViewController`. So how do we make sure changes in `ChecklistViewController` get saved now?

We could give `ChecklistViewController` a reference to the `AllListsViewController` and have it call its `saveChecklists` method whenever we change something, but that introduces a *child-parent dependency* and we've been trying hard to avoid those.

You may think: ah, we could use a delegate for this. True — and if you thought that indeed then I'm very proud — but instead we'll rethink our saving strategy.

Is it really necessary to save our changes all the time? While the app is running, the data model sits in working memory and is always up-to-date. We never have to load anything from the file (the long-term storage memory) because we did that already when the app started. From then on we always make the changes to our objects in the working memory. It is the file that becomes out-of-date, which is the reason we always saved our changes — to keep the file in sync with what we had in memory.

The reason we use the file is that we can restore our data model in working memory after the app gets terminated. But until that happens, the data in our working memory will do just fine. We just need to make sure that we save our data to the file just before the app gets terminated. In other words, the only time we save is when we actually need to keep our data safe. Not only is this more efficient, especially if we have a lot of data, it also is simpler to program. We no longer need to worry about saving every time the user makes a change to the data, only right before the app terminates.

There are three situations in which an app can terminate:

1. While the user is running the app. This doesn't happen very often anymore on iOS 4 and up, but earlier versions of iOS did not support multitasking apps. Receiving an incoming phone call, for example, would kill the currently running app. On iOS 4 and better the app will simply be suspended in the background when that happens. There are also situations where iOS may forcefully terminate a running app, for example if the app becomes unresponsive or runs out of memory.
2. When the app is suspended in the background. Most of the time iOS keeps these apps around for a long time. Their data is frozen in memory and no computations are taking place. (When you resume a suspended app, it literally continues from where it left off.) Sometimes the OS needs to make room for an app that requires a lot of working memory — often a game — and then it simply wipes the suspended apps from memory. The apps are not notified of this.
3. The app crashes. There are ways to detect crashes but handling them can be very tricky. Trying to deal with the crash may actually make things worse. The best way to avoid crashes is to make no programming mistakes! :-)

Fortunately for us, iOS will inform the app about significant changes such as: you are about to be terminated, and: you are about to be suspended. We can listen for these

events and save our data at that point. That will ensure our on-file representation of the data model is always up-to-date when the app does terminate.

The ideal place for handling these notifications is inside the *application delegate*. We haven't spend much time with this object before, but every app has one and as its name implies, it is the delegate object for notifications that concern the app as a whole. This is where you receive the "app will terminate" and "app will be suspended" notifications.

In fact, if you look inside ChecklistsAppDelegate.m, you'll see the methods:

```
- (void)applicationDidEnterBackground:(UIApplication *)application
```

and:

```
- (void)applicationWillTerminate:(UIApplication *)application
```

There are a few others, but these are the ones we need. (The Xcode template put helpful comments inside these methods, so you know what to do with them.)

Now the trick is, how do we call `AllListsViewController`'s `saveChecklist` method from these delegate methods? The app delegate does not know anything about `AllLists ViewController` yet. When you use a nib, it is easy to connect the view controller to an outlet in the App Delegate object. Unfortunately, Storyboards do not seem to have this feature at the moment.

We have to use some trickery to find the `AllListsViewController` from within the app delegate.

» At the top of ChecklistsAppDelegate.m, add:

ChecklistsAppDelegate.m

```
#import "AllListsViewController.h"
```

» Above `applicationDidEnterBackground`, add this new method:

ChecklistsAppDelegate.m

```
- (void)saveData
{
  UINavigationController *navigationController = (UINavigationController *)self↵
      .window.rootViewController;
```

```
  AllListsViewController *controller = (AllListsViewController *)[↵
      navigationController.viewControllers objectAtIndex:0];
  [controller saveChecklists];
}
```

» Change the `applicationDidEnterBackground` and `applicationWillTerminate` methods to:

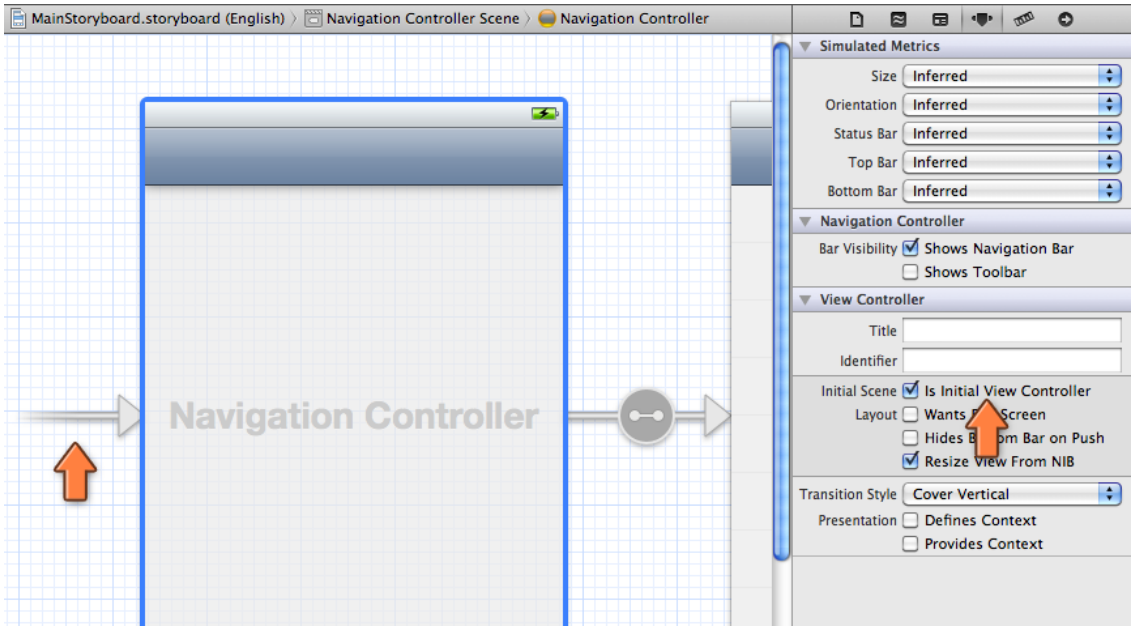
ChecklistsAppDelegate.m

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
  [self saveData];
}

- (void)applicationWillTerminate:(UIApplication *)application
{
  [self saveData];
}
```

The `saveData` method looks at the `self.window` property to find the `UIWindow` object that contains the storyboard. `UIWindow` is the top-level container for all your app's views. There is only one `UIWindow` object in your app (unlike desktop apps, which usually have multiple windows).

Normally you don't need to do anything with your `UIWindow`, but in this case we ask it for its `rootViewController`. This is the very first view controller from our storyboard, the navigation controller all the way over on the left. You can see this in the Storyboard editor because this navigation controller has the "Is Initial View Controller" flag set and a big arrow pointing at it:

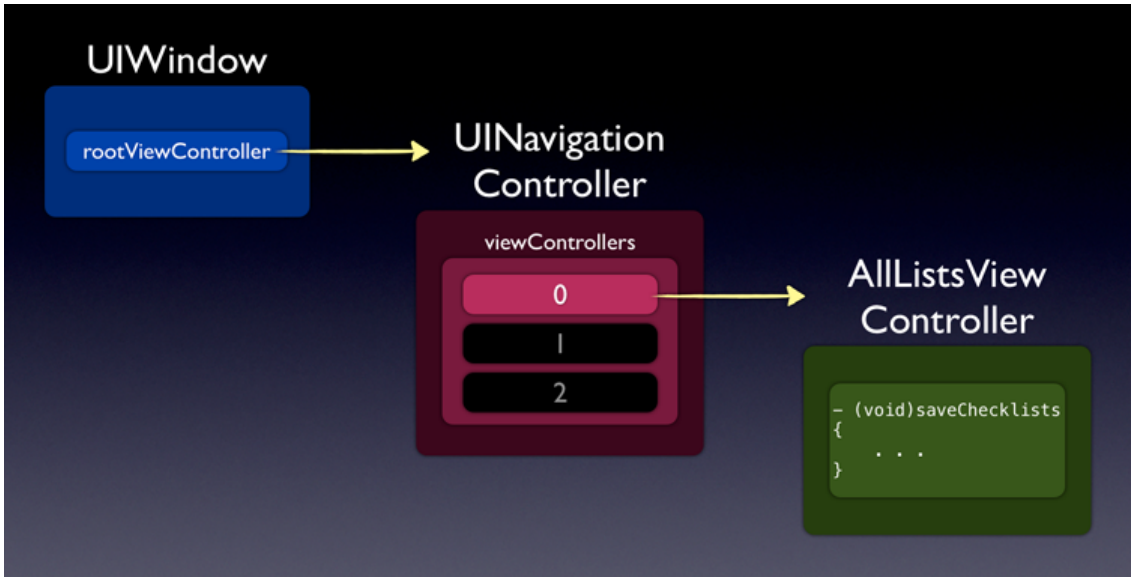
The left-most navigation controller is the window's root view controller



Once we have the navigation controller, we can find the `AllListsViewController` and then call its `saveChecklists` method. Unfortunately, the `UINavigationController` does not have a "rootViewController" property, so we have to look into its `viewControllers` array to find the bottom one.

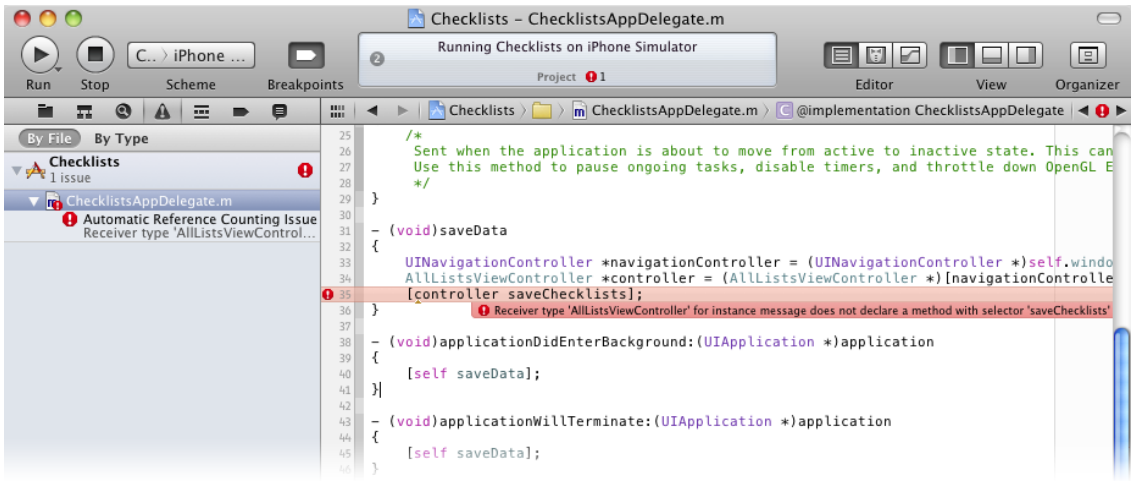
The `UINavigationController` does have a `topViewController` property but we cannot use it here: the "top" view controller is the screen that is currently displaying, which may very well be the `ChecklistViewController`. We don't want to send the `saveChecklists` message to that screen — it doesn't have a method to handle that message and the app will crash!

From the root view controller to the AllListsViewController



There is a small problem with the changes we've made so far: Xcode gives the error that "Receiver type 'AllListsViewController' for instance message does not declare a method with selector 'saveChecklists"' or the error "No visible @interface for 'AllListsViewController' declares the selector 'saveChecklists"'. How can this be when we've definitely added the `saveChecklists` method to the `AllListsViewController` object?

Xcode error: we're calling a method on AllListsViewController that it doesn't have



We haven't talked much about the distinction between interface and implementation yet, but what an object shows on the outside is different from what it has on the inside. That's done on purpose because its internals — the so-called *implementation details* — are not interesting to the user of the object. So we hide as much as possible inside the object and only show a few things on the outside.

For example, instance variables are necessary for the implementation only, not for the

users of the object. That's the reason we put them in the `@implementation` section and not in the `@interface` section.

Simply put, the .h file is the interface of an object and the .m file is its implementation.

We've only added the `saveChecklists` method to the AllListsViewController.m file, inside the `@implementation` section. That means it can be used only by this object itself and no one else. Other objects cannot see this method. It is usually a good idea to hide methods unless other objects need to be able to use them.

To make the `saveChecklists` method accessible to other objects, we need to add its *signature* to the .h file.

» Open AllListsViewController.h and add the line:

AllListsViewController.h

```
- (void)saveChecklists;
```

The complete .h file now looks like this:

AllListsViewController.h

```
#import <UIKit/UIKit.h>
#import "ListDetailViewController.h"

@interface AllListsViewController : UITableViewController <↵
    ListDetailViewControllerDelegate>

- (void)saveChecklists;

@end
```

Because we've added `saveChecklists` to the object's `@interface` section, other objects can now use it.

» Run the app, add some checklists, add items to those lists, set some checkmarks. Then press the Home button on the simulator to make the app go to the background. Look inside the app's Documents folder using Finder. There is now a new Checklists.plist file here. Press Stop in Xcode to terminate the app. Run the app again and your data should still be there. Awesome!

Xcode's Stop button

Important note: When you press Xcode's Stop button, the application delegate will not receive the `applicationWillTerminate` notification. Xcode kills the app without mercy. Therefore, to test the saving behavior, first tap the Home button to make the app go into the background and then press Stop. If you don't press Home first, you'll lose your data.

Improving the data model

The above code works but we can do a little better. We have made data model objects for `Checklist` and `ChecklistItem`, but there is still code in `AllListsViewController` — loading and saving the Checklists.plist object — that really belongs in the data model.

I prefer to create a top-level `DataModel` object for most of my apps. For this app, the `DataModel` object will contain the array of `Checklist` objects. We can move the code for loading and saving into this new `DataModel` object.

» Add a new file to the project, Objective-C class, "Subclass of NSObject". Save as "DataModel".

» Change DataModel.h to the following:

DataModel.h

```
#import <Foundation/Foundation.h>

@interface DataModel : NSObject

@property (nonatomic, strong) NSMutableArray *lists;

- (void)saveChecklists;

@end
```

We've added a `lists` property and the `saveChecklist` method. `DataModel` will be taking over these responsibilities from `AllListsViewController`.

» Inside DataModel.m, add a `@synthesize` statement below the `@implementation` line:

DataModel.m

```
@synthesize lists;
```

» Change the `init` method to:

DataModel.m

```
- (id)init
{
  if ((self = [super init])) {
    [self loadChecklists];
  }
  return self;
}
```

» Add the following above `init`:

DataModel.m

```
- (NSString *)documentsDirectory
{
  NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, ↵
      NSUserDomainMask, YES);
  NSString *documentsDirectory = [paths objectAtIndex:0];
  return documentsDirectory;
}

- (NSString *)dataFilePath
{
  return [[self documentsDirectory] stringByAppendingPathComponent:@"Checklists↵
      .plist"];
}

- (void)saveChecklists
{
  NSMutableData *data = [[NSMutableData alloc] init];
  NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc] ↵
      initForWritingWithMutableData:data];
  [archiver encodeObject:self.lists forKey:@"Checklists"];
  [archiver finishEncoding];
  [data writeToFile:[self dataFilePath] atomically:YES];
}

- (void)loadChecklists
{
  NSString *path = [self dataFilePath];
  if ([[NSFileManager defaultManager] fileExistsAtPath:path]) {
    NSData *data = [[NSData alloc] initWithContentsOfFile:path];
    NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc] ↵
        initForReadingWithData:data];
    self.lists = [unarchiver decodeObjectForKey:@"Checklists"];
    [unarchiver finishDecoding];
  } else {
```

```
    self.lists = [[NSMutableArray alloc] initWithCapacity:20];
  }
}
```

I simply cut these lines out of `AllListsViewController`, so make sure they are no longer in that file. Also note that `lists` is now a property, so we should access it as `self.lists`.

» Change AllListsViewController.h to:

AllListsViewController.h

```
#import <UIKit/UIKit.h>
#import "ListDetailViewController.h"
#import "DataModel.h"

@interface AllListsViewController : UITableViewController <↵
    ListDetailViewControllerDelegate>

@property (nonatomic, strong) DataModel *dataModel;

@end
```

» We've removed the `saveChecklists` method and added the `dataModel` property.

You should already have removed the `documentsDirectory`, `dataFilePath`, `saveChecklists` and `loadChecklists` methods from AllListsViewController.m.

» Also remove the `lists` ivar.

» Synthesize the new `dataModel` property:

AllListsViewController.m

```
@synthesize dataModel;
```

» Change `initWithCoder` to the following:

AllListsViewController.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    self.dataModel = [[DataModel alloc] init];
  }
  return self;
```

```
}
```

This method creates the `DataModel` object.

We can no longer reference the `lists` ivar directly, because it no longer exists. Instead, we'll have to ask the `DataModel` for its `lists` property.

» Make the following changes:

AllListsViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
{
  return [self.dataModel.lists count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  . . .

  Checklist *checklist = [self.dataModel.lists objectAtIndex:indexPath.row];

  . . .
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
     *)indexPath
{
  Checklist *checklist = [self.dataModel.lists objectAtIndex:indexPath.row];

  . . .
}

- (void)tableView:(UITableView *)tableView commitEditingStyle:(↵
    UITableViewCellEditingStyle)editingStyle forRowAtIndexPath:(NSIndexPath *)↵
    indexPath
{
  [self.dataModel.lists removeObjectAtIndex:indexPath.row];

  . . .
}

- (void)tableView:(UITableView *)tableView ↵
    accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
{
  . . .
```

```
  Checklist *checklist = [self.dataModel.lists objectAtIndex:indexPath.row];

  . . .
}

- (void)listDetailViewController:(ListDetailViewController *)controller ↵
    didFinishAddingChecklist:(Checklist *)checklist
{
  int newRowIndex = [self.dataModel.lists count];
  [self.dataModel.lists addObject:checklist];

  . . .
}

- (void)listDetailViewController:(ListDetailViewController *)controller ↵
    didFinishEditingChecklist:(Checklist *)checklist
{
  int index = [self.dataModel.lists indexOfObject:checklist];

  . . .
}
```

» One last change, in ChecklistsAppDelegate.m:

ChecklistsAppDelegate.m

```
- (void)saveData
{
  UINavigationController *navigationController = (UINavigationController *)self↵
      .window.rootViewController;
  AllListsViewController *controller = (AllListsViewController *)[↵
      navigationController.viewControllers objectAtIndex:0];
  [controller.dataModel saveChecklists];
}
```

We no longer call `saveChecklists` on `AllListsViewController` but on its `dataModel` property.

» Do a clean build (Product → Clean) and run the app. Verify that everything still works.

You can find the project files for the app up to this point under “08 - Improved Data Model” in the tutorial’s Source Code folder.

Using NSUserDefaults to remember stuff

We now have an app that lets you create checklists and add to-do items to those lists.All of this data is saved to long-term storage so even if the app gets terminated, nothing is lost. There are some user interface improvements we can make, though.

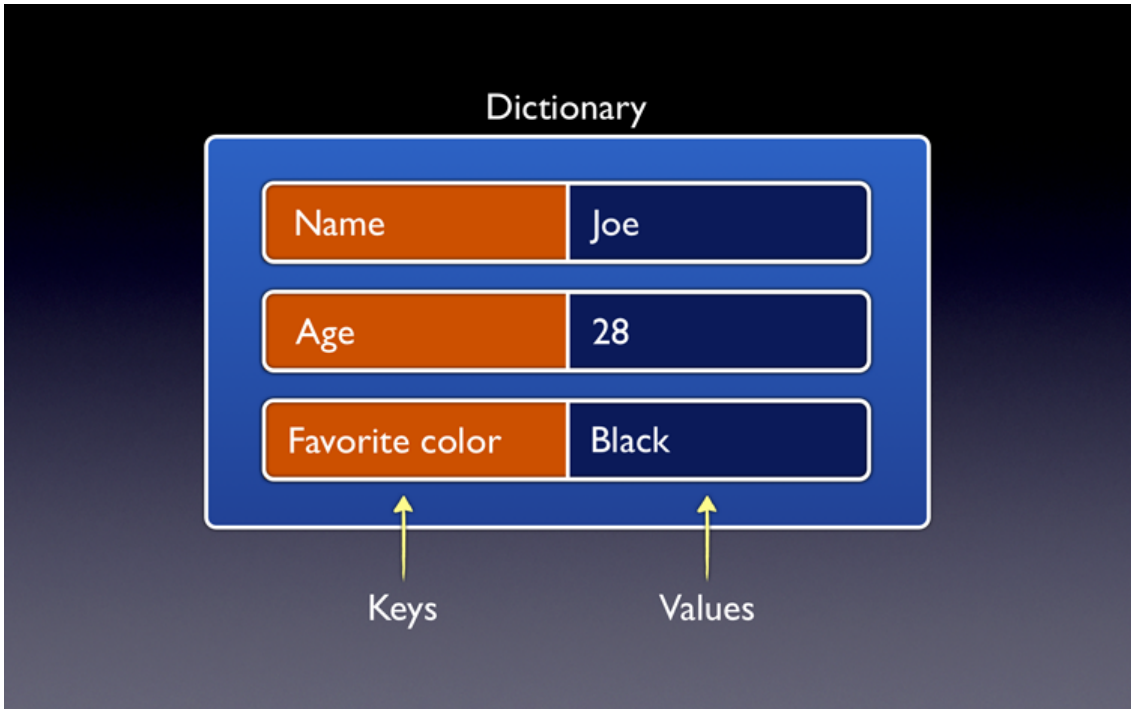
Imagine the user is on the Birthdays checklist and presses the Home button to switch to another app. The Checklists app is now suspended. Suppose that at some point the app gets terminated. When the user reopens the app it no longer is on Birthdays but on the main screen. Because it was terminated the app didn't simply resume where it left off, but got launched anew.

You might be able to get away with this, as apps don't get terminated often (unless you play a lot of games that eat up memory) but little things like this matter in iOS apps. Fortunately, it's fairly easy to remember whether the user has opened a checklist and to switch to it when the app starts up.

We could store this information in the Checklists.plist file, but especially for simple settings such as this there is the `NSUserDefaults` object.

`NSUserDefaults` works like a dictionary, which is a collection object for storing key-value pairs. We've already seen the array collection, which stores an ordered list of objects. The dictionary is another very common collection that looks like this:

A dictionary is a collection of key-value pairs



Dictionaries in Objective-C are handled by the `NSDictionary` and `NSMutableDictionary` objects. You can put objects into the dictionary under a reference key and then retrieve it later using that key. This is, in fact, how Info.plist works. This plist file is read into a dictionary and then iOS uses the various keys (on the left hand) to obtain the values (on the right hand). Keys are usually strings but values can be any type of object.

`NSUserDefaults` isn't a true dictionary, but it acts like one. When you insert new values into `NSUserDefaults`, they are saved somewhere in your app's sandbox so these values persist even after the app terminates. You don't want to store huge amounts of data inside `NSUserDefaults`, but it's ideal for small things like settings — and for remembering what screen the app was on when it closed.

This is what we are going to do:

- On the segue from the main screen (`AllListsViewController`) to the checklist screen (`ChecklistViewController`), we write the row index of the selected checklist into `NSUserDefaults`. This is how we'll remember which checklist was selected. We could have saved the name of the checklist instead of the row index, but what would happen then if two checklists have the same name? Unlikely, but not impossible. Using the row index guarantees that we'll always select the proper one.
- When the user presses the back button to return to the main screen, we have to remove this value from `NSUserDefaults` again. It is common to set a value such as this to -1 to mean "no value". Why -1? We start counting rows at 0, so we can't use 0 or a positive number (unless we use a huge number such as 1000000; it's very unlikely the user will make that many checklists). -1 is not a valid row index and because it's a negative value it looks weird, so that makes it easy to spot during debugging.
- If the app starts up and the value from `NSUserDefaults` isn't -1, then the user was previously viewing the contents of a checklist and we have to manually perform a segue to the `ChecklistViewController` for the corresponding row.

Phew, it's more work to explain this in English than writing the actual code. ;-)

Let's start with the segue from the main screen.

» In AllListsViewController.m, change `didSelectRowAtIndexPath` to the following:

AllListsViewController.m

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
```

```
  [[NSUserDefaults standardUserDefaults] setInteger:indexPath.row forKey:@"↵
      ChecklistIndex"];

  Checklist *checklist = [self.dataModel.lists objectAtIndex:indexPath.row];
  [self performSegueWithIdentifier:@"ShowChecklist" sender:checklist];
}
```

In addition to what this method did before, we now store the index of the selected row into `NSUserDefaults` under the key "ChecklistIndex".

To recognize whether the user presses the back button on the navigation bar, we have to set a delegate for the navigation controller. The logical place for this delegate is the `AllListsViewController`.

» Add the delegate protocol to the `AllListsViewController` `@interface` line:

AllListsViewController.h

```
@interface AllListsViewController : UITableViewController <↵
    ListDetailViewControllerDelegate, UINavigationControllerDelegate>
```

As you can see, a view controller can be a delegate for many other objects at once. `AllListsViewController` is now the delegate for both the `ListDetailViewController` and the `UINavigationController`, but also implicitly for the `UITableView` (because it is a table view controller).

» Add the delegate method to the bottom of AllListsViewController.m:

AllListsViewController.m

```
- (void)navigationController:(UINavigationController *)navigationController ↵
    willShowViewController:(UIViewController *)viewController animated:(BOOL)↵
    animated
{
  if (viewController == self) {
    [[NSUserDefaults standardUserDefaults] setInteger:-1 forKey:@"↵
        ChecklistIndex"];
  }
}
```

This method is called whenever the navigation controller will slide to a new screen. If the back button was pressed, then the new view controller is the `AllListsViewController` itself and we set the "ChecklistIndex" value in `NSUserDefaults` to -1.

The only thing that remains is to check at startup which checklist we need to show and then perform the segue manually. We'll do that in `viewDidAppear`.

» Change (or add) the `viewDidAppear` method:

AllListsViewController.m

```
- (void)viewDidAppear:(BOOL)animated
{
  [super viewDidAppear:animated];

  self.navigationController.delegate = self;

  int index = [[NSUserDefaults standardUserDefaults] integerForKey:@"↵
      ChecklistIndex"];
  if (index != -1) {
    Checklist *checklist = [self.dataModel.lists objectAtIndex:index];
    [self performSegueWithIdentifier:@"ShowChecklist" sender:checklist];
  }
}
```

This method is called after the view controller has become visible. We make ourselves the delegate for the navigation controller and then check `NSUserDefaults` whether we have to perform the segue.

If the value of the "ChecklistIndex" setting is not -1, then the user was previously viewing a checklist and we should segue to that screen. As before, we send the `Checklist` object along with the `sender` parameter of `performSegueWithIdentifier`.

Note that `!=` means: not equal. It is the opposite of the == operator. (Some languages use <> for not equal but that won't work in Objective-C.)

I'm actually guilty of a bit of trickery here. `viewDidAppear` isn't just called when the app starts up but also every time the navigation controller slides the main screen back into view. Checking whether to restore the checklist screen needs to happen just once when the app starts, so why did I put this logic in `viewDidAppear`?

The very first time that `AllListsViewController`'s screen becomes visible we do not want the `willShowViewController` delegate method to be called, as that would always overwrite the old value of "ChecklistIndex" with -1, before we've had a chance to restore the old screen. By waiting to register `AllListsViewController` as the navigation controller delegate until it is visible, we avoid this problem.

When the user presses the back button, the navigation controller will call `willShowViewController` before `viewDidAppear`. Because the value of "ChecklistIndex" will now always be -1, `viewDidAppear` does not trigger a segue again. There are other ways to solve this particular issue but this approach is simple, so I like it.

Is all of this going way over your head? Don't worry about it. Let it sink in and soon enough it will all start to make sense. Even better, probe around in the code. Change things around to see what the effect is. That's the quickest way to learn!

» Run the app and go to a checklist screen. Press the simulator's Home button, followed by Stop to quit the app.

Tip: You need to press the Home button because `NSUserDefaults` may not immediately save its settings to disk and therefore you may lose your changes if you kill the app from within Xcode.

» Run the app again and you'll notice that Xcode immediately switches to the screen where you were last at. Cool, huh!

» Now do the following: Stop the app and delete it from the Simulator. You can either reset the whole simulator from its menu (iOS Simulator → Reset Contents and Settings) or hold down the app icon until it starts to wiggle and then delete it just as you would on your iPhone.

Then run the app again from within Xcode and watch it crash:

```
*** Terminating app due to uncaught exception 'NSRangeException', reason: '***
-[__NSArrayM objectAtIndex:]: index 0 beyond bounds for empty array'
```

The app crashes in `viewDidAppear` on the line:

```
Checklist *checklist = [self.dataModel.lists objectAtIndex:index];
```

What's going on here? Apparently the value of the `index` variable is 0, even though there should be nothing in `NSUserDefaults` yet because this is a fresh install of our app. We didn't write anything in the "ChecklistIndex" key yet.

It turns out that `NSUserDefaults`'s `integerForKey` method returns 0 if it cannot find the value for the key you specify, but in our app 0 is a valid row index. At this point the app

doesn't have any checklists yet, so index 0 does not exist in the `lists` array. That is why the app crashes.

What we would like instead, is that `NSUserDefaults` returns -1 if the "ChecklistIndex" key isn't set, because to us -1 means: show the main screen instead of a specific checklist. Fortunately, `NSUserDefaults` will let us set default values for the default values. Yep, you read that correctly.

Let's do that in our `DataModel` object.

» Add the following method above `init` inside DataModel.m:

DataModel.m

```
- (void)registerDefaults
{
  NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt:-1], @"ChecklistIndex",
    nil];

  [[NSUserDefaults standardUserDefaults] registerDefaults:dictionary];
}
```

This creates a new `NSDictionary` object and adds the value -1 for the key "ChecklistIndex". `NSUserDefaults` will use the values from this dictionary if you ask it for a key but it cannot find anything under that key.

» Change `init` to call this new method:

DataModel.m

```
- (id)init
{
  if ((self = [super init])) {
    [self loadChecklists];
    [self registerDefaults];
  }
  return self;
}
```

» Run the app again and now it should no longer crash.

Why did we do this in `DataModel`? Well, I don't really like to sprinkle all of these calls to `NSUserDefaults` throughout the code. In fact, lets move all of the `NSUserDefaults` stuff into `DataModel`.

» Add the following methods to the bottom of DataModel.m, before `@end`:

DataModel.m

```
- (int)indexOfSelectedChecklist
{
  return [[NSUserDefaults standardUserDefaults] integerForKey:@"ChecklistIndex"↵
      ];
}

- (void)setIndexOfSelectedChecklist:(int)index
{
  [[NSUserDefaults standardUserDefaults] setInteger:index forKey:@"↵
      ChecklistIndex"];
}
```

We're doing this so the rest of the code won't have to worry about `NSUserDefaults`. Our other objects just have to call the proper methods on `DataModel`. Hiding implementation details is an important Object-Oriented Programming principle. If we decide later that we want to store these settings somewhere else, for example in a database, then we only have to change this in one place, in `DataModel`. The rest of the code will be oblivious to these changes and that's a good thing.

We need to add these methods to DataModel.h too, otherwise the other objects cannot use them.

» Add the method signatures to DataModel.h:

DataModel.h

```
- (int)indexOfSelectedChecklist;
- (void)setIndexOfSelectedChecklist:(int)index;
```

» Update the code in AllListsViewController.m to use these new methods:

AllListsViewController.m

```
- (void)viewDidAppear:(BOOL)animated
{
    . . .

  int index = [self.dataModel indexOfSelectedChecklist];

  . . .
}
```

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
  [self.dataModel setIndexOfSelectedChecklist:indexPath.row];

  . . .
}

- (void)navigationController:(UINavigationController *)navigationController ↵
    willShowViewController:(UIViewController *)viewController animated:(BOOL)↵
    animated
{
  if (viewController == self) {
    [self.dataModel setIndexOfSelectedChecklist:-1];
  }
}
```

» Run the app again and make sure everything still works.

It's pretty nice that the app now remembers what screen you were on, but this new feature has also introduced a subtle bug in the app. Here's how to reproduce it:

» Start the app and add a new checklist. Also add a new to-do item to this list. Now kill the app from within Xcode.

Because you did not press the Home button, the new checklist and its item were not saved to Checklists.plist. However, there is a (small) chance that `NSUserDefaults` did save its changes to disk and now thinks this new list is selected. That's a problem because that list doesn't exist anymore (it never made it into Checklists.plist).

`NSUserDefaults` will save its changes at indeterminate times so it could have saved before you terminated the app. (This is especially true when you implement local notifications later in this tutorial, where we force `NSUserDefaults` to save its changes every time you add a new to-do item. Then the app is guaranteed to crash at this point.)

» Run the app again and — if you're lucky? — it will crash with:

```
Checklists[1124:707] *** Terminating app due to uncaught exception
'NSRangeException', reason: '*** -[__NSArrayM objectAtIndex:]:
index 1 beyond bounds [0 .. 0]'
```

The problem is that `NSUserDefaults` and the contents of Checklists.plist are out-of-sync. `NSUserDefaults` thinks we need to select a checklist that doesn't actually exist. Every time you run the app it will now crash. Yikes!

This situation shouldn't really happen during regular usage because we used the Xcode Stop button to kill the app. Under normal circumstances the user would press the Home button at some point and as the app goes into the background it will save both Checklists.plist and `NSUserDefaults` and everything is in sync again. However, the OS can always decide to terminate the app and then this situation could occur.

Even though there's only a small chance that this can go wrong in practice, we should really protect ourselves against it. These are the kinds of bug reports you don't want to get because often you have no idea what the user did to make it happen. It's good to stick to the practice of *defensive programming*, to check for boundary cases and be able to gracefully handle them even if they are unlikely to occur.

In our case, we can fix `AllListsViewController`'s `viewDidAppear` method to deal with this situation.

» Change `viewDidAppear` to:

AllListsViewController.m

```
- (void)viewDidAppear:(BOOL)animated
{
  [super viewDidAppear:animated];

  self.navigationController.delegate = self;

  int index = [self.dataModel indexOfSelectedChecklist];
  if (index >= 0 && index < [self.dataModel.lists count]) {
    Checklist *checklist = [self.dataModel.lists objectAtIndex:index];
    [self performSegueWithIdentifier:@"ShowChecklist" sender:checklist];
  }
}
```

The changed line is this:

```
  if (index >= 0 && index < [self.dataModel.lists count]) {
```

Instead of just checking for `index != 1`, we now do a more precise check to determine whether `index` is valid. It should be between 0 and the number of checklists in our data model. If not, then we simply don't segue. This will prevent `objectAtIndex` from asking for an object that doesn't exist.

We haven't seen the `&&` operator before. This symbol means "logical and". It is used as follows:

```
if (something && somethingElse) {
  // do stuff
}
```

This reads: if something is true *and* something else is also true, then do stuff.

In `viewDidAppear` we only perform the segue when `index` is 0 or greater *and also* less than the number of checklists, which means it's only valid if it lies in between those two values.

With this defensive check in place, we're guaranteed that the app will not try to segue to a checklist that doesn't exist, even if our data is out-of-sync.

Note that the app doesn't remember whether the user had the Add/Edit Checklist or Add/Edit Item screen open. These kinds of modal screens are supposed to be temporary. You open them to make a few changes and then close them again. If the app goes to the background and is terminated, then it's no big deal if the modal screen disappears.

At least that is true for this app. If you have an app that allows the user to make many complicated edits in a modal screen, then you may want to persist those changes when the app closes so the user won't lose all his work in case the app is killed.

The first-run experience

Let's use `NSUserDefaults` for something else. It would be nice if the first time you ran the app it created a default checklist for you, simply named "List", and switched the screen to that list. This enables you to start adding to-do items right away. That's how the standard Notes app works too: you can start typing a note right after launching the app for the very first time, but you can also go one level back in the navigation hierarchy to see a list of all notes.

To pull this off, we need to keep track in `NSUserDefaults` whether this is the first time the user runs the app. If it is, then we create a new `Checklist` object. We can do all of this inside `DataModel`.

» Add the following import to the top of DataModel.m:

DataModel.m

```
#import "Checklist.h"
```

We will add a new default to the `registerDefaults` method. The key for this value is "FirstTime".

» Change the `registerDefaults` method:

DataModel.m

```
- (void)registerDefaults
{
  NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt:-1], @"ChecklistIndex",
    [NSNumber numberWithBool:YES], @"FirstTime",
    nil];

  [[NSUserDefaults standardUserDefaults] registerDefaults:dictionary];
}
```

We say `[NSNumber numberWithBool:YES]` because we want the value of "FirstTime" to be `YES` if this is the first time we run the app after a fresh install.

Primitive values vs objects

Dictionaries cannot contain primitive values such as `int` and `BOOL`, only objects. The same thing goes for arrays. If you want to put an `int` or `BOOL` value into a dictionary or array, you have to convert it into an `NSNumber` object first.

I have briefly mentioned the difference between primitive datatypes and objects a few times before. In some programming languages everything is an object; in Objective-C *almost* everything is an object. There is a cost associated with using objects and for certain simple operations, such as doing arithmetic with whole numbers, it's easier and faster to do this with primitive values instead.

You can tell primitive types and objects apart by the `*` that follows their name. Only objects have this asterisk. In addition, you do not use `[[alloc] init]` to create primitive values.

Sometimes you need to convert between primitive types and objects. To put an `int` or `BOOL` value into a dictionary, you need to stuff it into an `NSNumber` object first. The other way around is possible too: to get an integer value out of an `NSNumber`, you'd do `[number intValue]`. For a `BOOL` that is `[boolValue]`.

If you're still confused about the difference between primitive values and objects, then rest assured, a more detailed explanation is forthcoming in the next tutorial.

» Add the `handleFirstTime` method, above `init`:

DataModel.m

```
- (void)handleFirstTime
{
  BOOL firstTime = [[NSUserDefaults standardUserDefaults] boolForKey:@"↵
      FirstTime"];
  if (firstTime) {
    Checklist *checklist = [[Checklist alloc] init];
    checklist.name = @"List";
    [self.lists addObject:checklist];
    [self setIndexOfSelectedChecklist:0];
    [[NSUserDefaults standardUserDefaults] setBool:NO forKey:@"FirstTime"];
  }
}
```

Here we check `NSUserDefaults` for the value of the "FirstTime" key. Interestingly enough, we can simply forget about the whole `NSNumber` thing and ask the `NSUserDefaults` directly for a boolean value. Converting to an `NSNumber` was only necessary when we registered the defaults. (That's because `NSUserDefaults` isn't a true dictionary object but only acts like one.)

If the "FirstTime" value is `YES`, then this is the first time the app is being run. We create a new `Checklist` object and add it to the array. We call `setIndexOfSelectedChecklist` to make sure we'll automatically segue to this new checklist in `AllListsViewController`'s `viewDidAppear`. Finally, we'll set "FirstTime" to `NO`, so this bit of code won't be executed again the next time the app starts up.

» Call this new method in `init`:

DataModel.m

```
- (id)init
{
  if ((self = [super init])) {
    [self loadChecklists];
    [self registerDefaults];
    [self handleFirstTime];
  }
  return self;
}
```

» Remove the app from the Simulator and run it again from Xcode. Because it's the first time we run the app (at least from the app's perspective), it will automatically create a new checklist named List and switch to it.

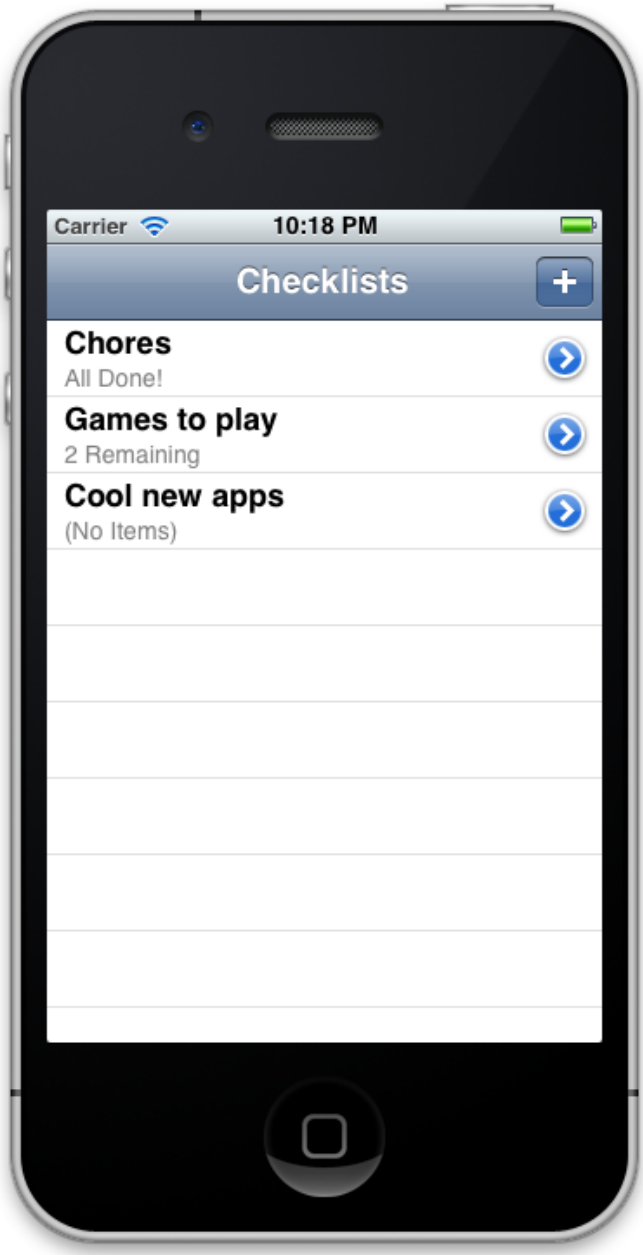
Improving the user experience

Showing the number of to-do items remaining

There are a few small features I'd like to add, just to polish the app a little more. After all, we're building a real app here — if you want to make top-notch apps, you have to pay attention to those details.

In the main screen, for each checklist I want to show the number of to-do items that do not have checkmarks yet:

Each checklist shows how many items are still left to-do



First, we need a way to count these items.

» Add the following method to Checklist.h:

Checklist.h

```
- (int)countUncheckedItems;
```

With this method we can ask any `Checklist` object how many of its `ChecklistItem` objects do not yet have their checkmark set. The method returns this count as an `int` value.

» Add the implementation of the `countUncheckedItems` method to Checklist.m:

Checklist.m

```
- (int)countUncheckedItems
{
  int count = 0;
  for (ChecklistItem *item in self.items) {
    if (!item.checked) {
      count += 1;
    }
  }
  return count;
}
```

This method loops through the `ChecklistItem` objects from the `items` array. If the `item` object has its `checked` property set to `NO`, we increment the local variable `count` by 1. When we've looked at all the objects, we return the value of this count to the caller.

Remember that the `!` operator negates the result. So it `item.checked` is `YES`, then `!item.checked` will make it `NO`. You should read it as "if not `item.checked`".

This also needs an import for `ChecklistItem` or Xcode won't let us call `item.checked` on the object.

» Add the import at the top of Checklist.m:

Checklist.m

```
#import "ChecklistItem.h"
```

» Go to AllListsViewController.m and change `cellForRowAtIndexPath` to:

AllListsViewController.m

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  static NSString *CellIdentifier = @"Cell";

  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:↵
      CellIdentifier];
  if (cell == nil) {
```

```
    cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle ↵
        reuseIdentifier:CellIdentifier];
  }

  Checklist *checklist = [self.dataModel.lists objectAtIndex:indexPath.row];

  cell.textLabel.text = checklist.name;
  cell.accessoryType = UITableViewCellAccessoryDetailDisclosureButton;
  cell.detailTextLabel.text = [NSString stringWithFormat:@"%d Remaining", [↵
      checklist countUncheckedItems]];
  return cell;
}
```

Most of the code stays the same, except we now use `UITableViewCellStyleSubtitle` instead of `UITableViewCellStyleDefault`. The "subtitle" cell style adds a smaller, gray label below the main label. You can use the `detailTextLabel` property to access this subtitle label.

We call the `countUncheckedItems` method on the `Checklist` object and put the count into a new string that we place into the `detailTextLabel`.

» Run the app. For each checklist it will now show how many items still remain to be done.

One problem: The to-do count never changes. If you toggle a checkmark on or off, or add new items, the "to do" count remains the same. That's because we create these table view cells once and never update their labels.

Exercise: Think of all the situations that will cause this "still to do" count to change. ■

Answer:

- The user toggles a checkmark on an item. When the checkmark is set, the count goes down. When the checkmark is removed, the count goes up again.
- The user adds a new item. New items don't have their checkmark set, so adding a new item should increment the count.
- The user deletes an item. The count should go down but only if that item had no checkmark.

These changes all happen in the `ChecklistViewController` but the "still to do" label is shown in the `AllListsViewController`. So how do we let the `AllListsViewController` know about this?

If you thought, use a delegate, then you're starting to get the hang of this. We could make a new `ChecklistViewControllerDelegate` protocol that sends messages when the following things happen:

- the user toggles a checkmark on an item
- the user adds a new item
- the user deletes an item

But what would the delegate — which would be `AllListsViewController` — do in return? It would simply set a new text on the cell's `detailTextLabel` in all cases.

This approach sounds good, only we're going to cheat and not use a delegate at all.

» Go to AllListsViewController.m and update (or add) the `viewWillAppear` method to do the following:

AllListsViewController.m

```
- (void)viewWillAppear:(BOOL)animated
{
  [super viewWillAppear:animated];

  [self.tableView reloadData];
}
```

Don't confuse this method with `viewDidAppear`. The difference is in the verb: *will* versus *did*. `viewWillAppear` is called before `viewDidAppear`.

The iOS API often does this: there is a "will" method that is invoked before something happens and a "did" method that is invoked after that something happened. Sometimes you need to do things before, sometimes after, and having two methods gives you the ability to choose whichever situation works best for you.

API (ay-pee-eye) stands for Application Programming Interface. When people say "the iOS API" they mean all the frameworks, objects, protocols and functions that are provided by iOS that you as a programmer can use to write apps. The iOS API consists of everything from UIKit, Foundation, Core Graphics, and so on. When people talk about "the Facebook API" or "the Google API", then they mean the services that these companies provide that allow you to write apps for those platforms.

In our case, we will simply tell the table view to reload its entire contents. That will cause `cellForRowAtIndexPath` to be called again for every visible row.

When you tap the back button on the `ChecklistViewController`'s navigation bar, the `AllListsViewController` screen will slide back into view. Just before that happens, `viewWillAppear` is called and thanks to the call to `reloadData` we will update all of the table cells, including the `detailTextLabels`.

Reloading all of the cells may be a little overkill but in this situation we can get away with it. It's unlikely the All Lists screen will contain many rows so reloading them is quite fast. And it saves us some work of having to make yet another delegate. Sometimes a delegate is the best solution, sometimes you can simply reload the entire table.

» Run the app and test that it works!

Exercise: Change the label to read "All Done!" when there are no more to-do items left to check. ■

Answer: Change the relevant code in `cellForRowAtIndexPath` to:

AllListsViewController.m

```
int count = [checklist countUncheckedItems];
if (count == 0) {
  cell.detailTextLabel.text = @"All Done!";
} else {
  cell.detailTextLabel.text = [NSString stringWithFormat:@"%d Remaining", ↵
      count];
}
```

Exercise: Now update the label to say "No Items" when the list is empty. ■

Answer:

AllListsViewController.m

```
int count = [checklist countUncheckedItems];
if ([checklist.items count] == 0) {
  cell.detailTextLabel.text = @"(No Items)";
} else if (count == 0) {
  cell.detailTextLabel.text = @"All Done!";
} else {
  cell.detailTextLabel.text = [NSString stringWithFormat:@"%d Remaining", ↵
      count];
}
```

Sorting the lists

Another thing you often need to do with lists is sort them in some particular order. Let's sort the list of checklists by name. Currently when you add a new checklist it is always appended to the end of the list.

Before we figure out how to sort an array, let's think about when we need to perform this sort:

- When a new checklist is added
- When a checklist is renamed

There is no need to re-sort when a checklist is deleted because that doesn't have any impact on the order of the other objects.

Currently we handle these two situations in `AllListsViewController`'s implementation of `didFinishAddingChecklist` and `didFinishEditingChecklist`.

» Change these methods to the following:

AllListsViewController.m

```
- (void)listDetailViewController:(ListDetailViewController *)controller ↵
    didFinishAddingChecklist:(Checklist *)checklist
{
  [self.dataModel.lists addObject:checklist];
  [self.dataModel sortChecklists];
  [self.tableView reloadData];

  [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)listDetailViewController:(ListDetailViewController *)controller ↵
    didFinishEditingChecklist:(Checklist *)checklist
{
  [self.dataModel sortChecklists];
  [self.tableView reloadData];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

We were able to remove a whole bunch of stuff from both methods because we now always do `reloadData` on the table view. It is no longer necessary to insert the new row manually, or to update the cell's `textLabel`. Instead we simply call `reloadData` to refresh the entire table's contents.

Again, we can get away with this because the table will only hold a handful of rows. If this table held hundreds of rows, a more advanced approach might be necessary. (You could figure out where the new or renamed `Checklist` object should be inserted and just update that row.)

The `sortChecklists` method on `DataModel` is new.

» Add its signature to DataModel.h:

DataModel.h

```
- (void)sortChecklists;
```

» And the full implementation in DataModel.m:

DataModel.m

```
- (void)sortChecklists
{
  [self.lists sortUsingSelector:@selector(compare:)];
}
```

`NSMutableArray` has a `sortUsingSelector` method that is really easy to use. A *selector* is the name of a method. Here we tell the `lists` array that it should be sorted using the `compare:` method. This method is not defined on the array itself but on the objects it contains, the `Checklists`.

The sort algorithm will call `[Checklist compare]` to see how the `Checklist` objects relate to one another. Because the sorting algorithm doesn't really know anything about our `Checklist` objects — or what it means for one `Checklist` to come before another — we have to help it out by providing this method.

» Add the `compare` method to Checklist.m:

Checklist.m

```
- (NSComparisonResult)compare:(Checklist *)otherChecklist
{
  return [self.name localizedStandardCompare:otherChecklist.name];
}
```

That's all we need to do. We're only looking at the name of this `Checklist` object versus the name of the `otherChecklist` object. The `name` property is an `NSString`, which already has a very convenient comparison method, `localizedStandardCompare`.

This method will compare the two `name` objects while ignoring lowercase vs uppercase (so "a" and "A" are considered equal) and taking into consideration the rules of the current locale. A *locale* is an object that knows about country and language-specific rules. Sorting in German may be different than sorting in English, for example.

So `NSMutableArray`'s `sortWithSelector` method will repeatedly ask one `Checklist` object how it compares to another `Checklist` object and then shuffle them around until the array is sorted. Inside that `Checklist` object's `compare` method, we simply compare the names of the two objects. If you wanted to sort on other criteria all you have to do is change the compare method.

Dynamic method name resolution (using selectors)

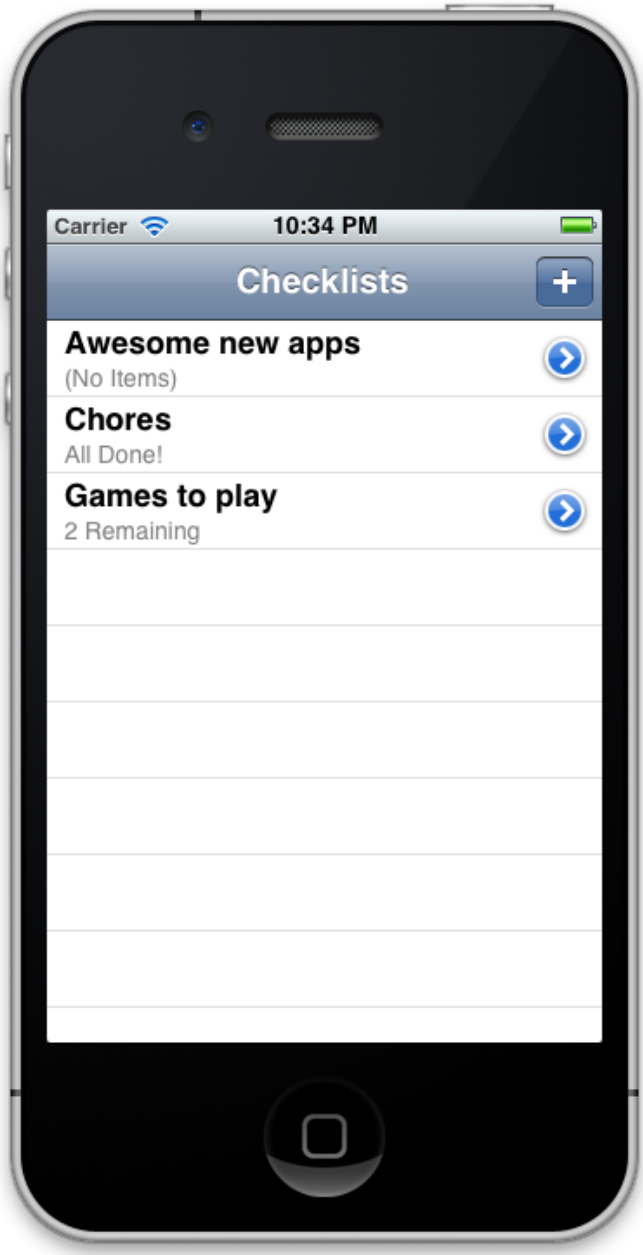
If you're wondering why we didn't add the signature for the `compare:` method to Checklist.h, then here's a little secret: you could do this but it's not necessary.

If we were calling `[checklist compare]` directly in our code then we would indeed need to declare this method in the Checklist.h file. But here we're using a selector, which will resolve the method name at *runtime* (i.e. when the app is running in the Simulator or on the device) rather than at compile-time.

For this type of dynamic method name resolution you don't need to add methods to the .h file, as the .h file isn't used for this. It's also a little more dangerous: you can call a selector on an object that doesn't exist. We've already seen that this makes the app crash with an "unrecognized selector sent to instance xxx" error message.

» Run the app and add some new checklists. Change their names and notice that the list is always sorted alphabetically.

New checklists are always sorted alphabetically

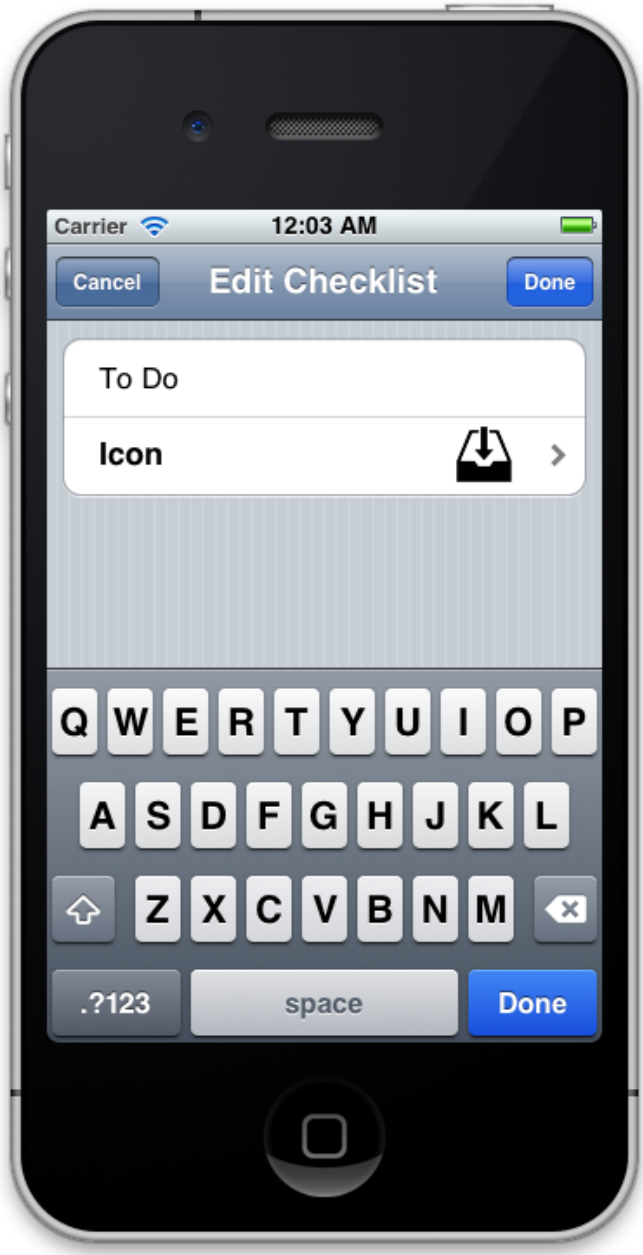


Adding icons to the checklists

Because I can't get enough of view controllers and delegates, let's add a new setting to the `Checklist` object that lets you choose an icon. I really want to cement these principles in your mind.

When we're done, the Add/Edit Checklist screen will look like this:

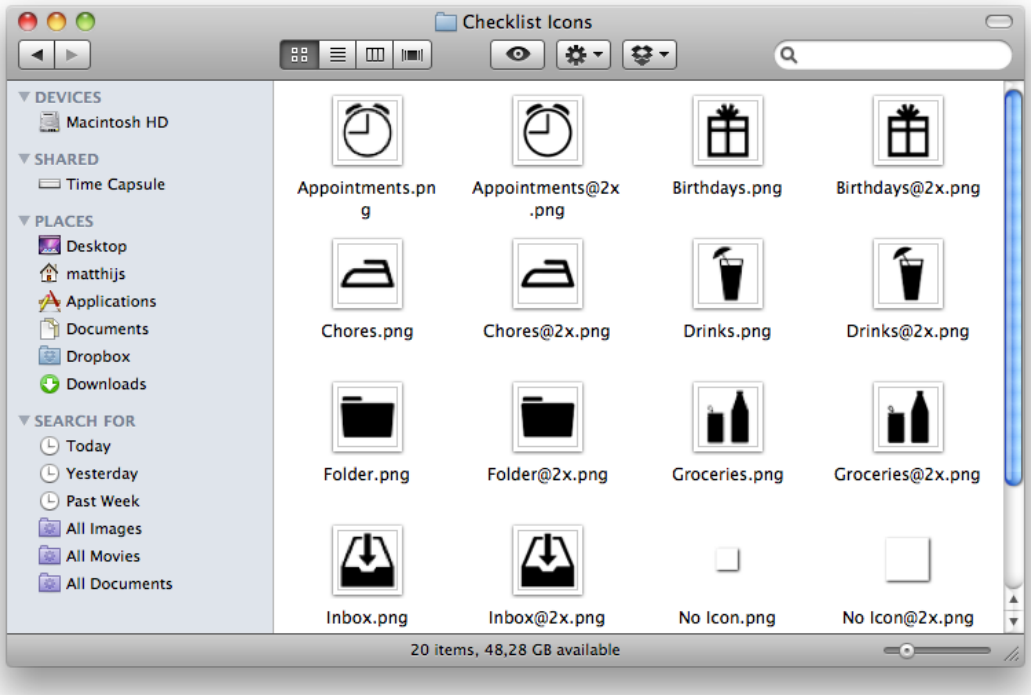
You can assign an icon to a checklist



We will add a row to the Add/Edit Checklist screen that opens a new screen that lets you pick an icon. This icon picker is a new view controller. We won't show it modally this time but push it on the navigation stack so it slides into the screen.

The Resources folder for this tutorial contains a folder named "Checklist Icons" with a selection of PNG images that depict different categories.

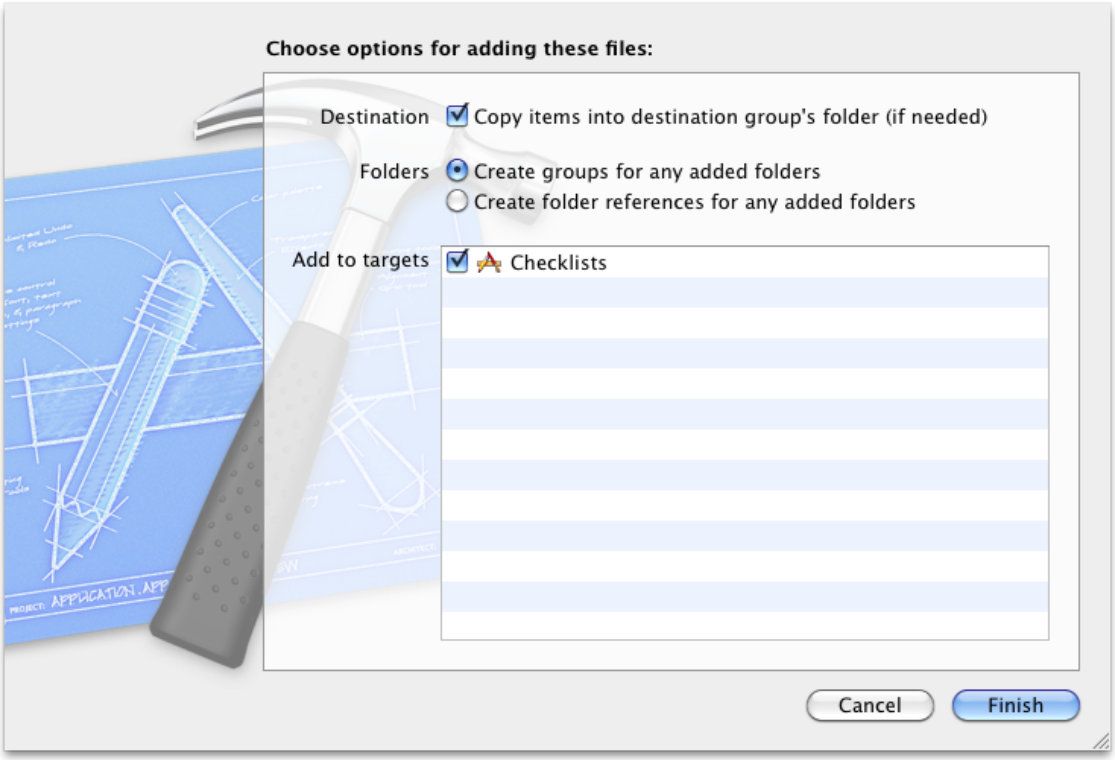
The various checklist icon images



» Add this entire folder to the project. You can either right-click the project name in the Project Navigator and choose “Add Files to Checklists” or simply drag the entire folder from Finder into the Project Navigator.

Make sure the “Copy items into destination group’s folder (if needed)” option is checked.

Copy the icon files into the project



» Add the following property to Checklist.h:

Checklist.h

```
@property (nonatomic, copy) NSString *iconName;
```

» Inside Checklist.m, add a line to synthesize the property:

Checklist.m

```
@synthesize iconName;
```

» Extend `initWithCoder` and `encodeWithCoder` to respectively load and save this icon name in the Checklists.plist file:

Checklist.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super init])) {
    self.name = [aDecoder decodeObjectForKey:@"Name"];
    self.items = [aDecoder decodeObjectForKey:@"Items"];
```

```
    self.iconName = [aDecoder decodeObjectForKey:@"IconName"];
  }
  return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
  [aCoder encodeObject:self.name forKey:@"Name"];
  [aCoder encodeObject:self.items forKey:@"Items"];
  [aCoder encodeObject:self.iconName forKey:@"IconName"];
}
```

» Just for testing, update the `init` method to the following:

Checklist.m

```
- (id)init
{
  if ((self = [super init])) {
    self.items = [[NSMutableArray alloc] initWithCapacity:20];
    self.iconName = @"Appointments";
  }
  return self;
}
```

This will give all checklists the “Appointments” icon. At this point we just want to see that we can make an icon — any icon — show up in the table view. When that works we can worry about letting the user pick the icon.

» Change `cellForRowAtIndexPath` in AllListsViewController.m to put the icon into the table view cell:

AllListsViewController.m

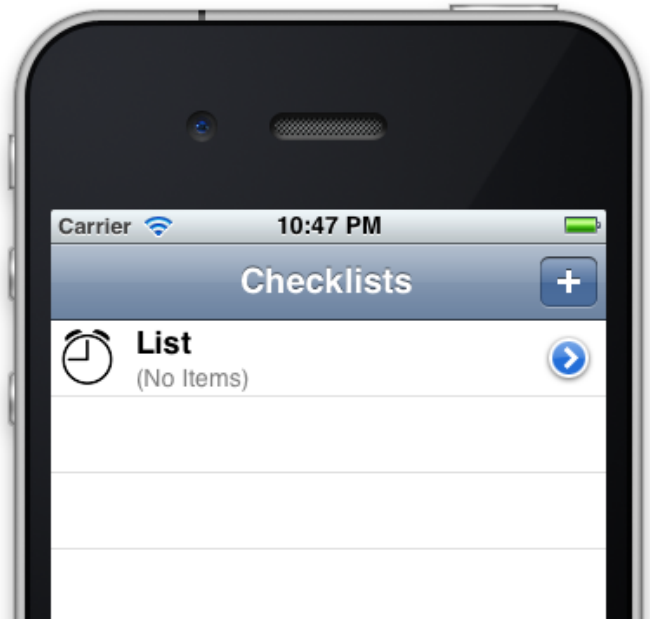
```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  . . .

  cell.imageView.image = [UIImage imageNamed:checklist.iconName];

  return cell;
}
```

The standard cell type we’re using (`UITableViewCellStyleSubtitle`) comes with a built-in `UIImageView` on the left. We can simply put our image into it. Easy peasy.

» Before running the app, remove the Checklists.plist file or uninstall the app from the Simulator because we've modified the file format again. We don't want weird crashes…

» Run the app and now each checklist should have a watch icon in front of its name. This will also look good in Retina mode because we supplied a @2x image as well.

The checklist now has an icon



Satisfied that this works, we can now change `Checklist`'s `init` to give each `Checklist` object an icon named "No Icon" by default.

» Change the `init` method to:

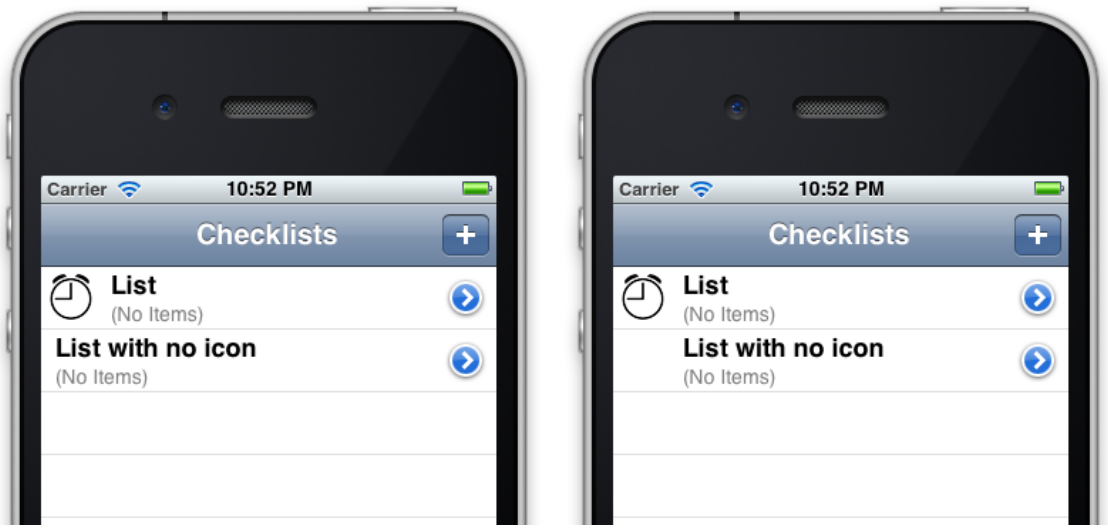
Checklist.m

```
- (id)init
{
  if ((self = [super init])) {
    self.items = [[NSMutableArray alloc] initWithCapacity:20];
    self.iconName = @"No Icon";
  }
  return self;
}
```

The "No Icon" image is a fully transparent PNG image with the same dimensions as the other icons. Using a transparent image is necessary to make all the checklists line up properly, even if they have no icon.

If we were to set `self.iconName` to `nil` instead, then the image view in the table view cell would remain empty and the text would align with the left margin of the screen. But that looks bad when other cells do have icons:

We use an empty image to properly align the text labels (right)



Let's add the icon picker screen. Add a new file for a `UITableViewController` subclass to the project. Name it "IconPickerViewController".

» Change IconPickerViewController.h to:

IconPickerViewController.h

```
#import <UIKit/UIKit.h>

@class IconPickerViewController;

@protocol IconPickerViewControllerDelegate <NSObject>
- (void)iconPicker:(IconPickerViewController *)picker didPickIcon:(NSString *)↵
    iconName;
@end

@interface IconPickerViewController : UITableViewController

@property (nonatomic, weak) id <IconPickerViewControllerDelegate> delegate;

@end
```

» In IconPickerViewController.m, change the top of the file to:

IconPickerViewController.m

```
@implementation IconPickerViewController {
  NSArray *icons;
}

@synthesize delegate;
```

» Also change `viewDidLoad`:

IconPickerViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  icons = [NSArray arrayWithObjects:
    @"No Icon",
    @"Appointments",
    @"Birthdays",
    @"Chores",
    @"Drinks",
    @"Folder",
    @"Groceries",
    @"Inbox",
    @"Photos",
    @"Trips",
    nil];
}
```

The instance variable `icons` is an `NSArray` that contains a list of icon names. These strings are both the text we will show on the screen and the name of the PNG file. The `icons` array is the data model for this table view. Note that it is a non-mutable `NSArray` (instead of an `NSMutableArray`) because the user cannot add or delete icons.

Convenience constructors

Earlier you've seen that new objects are created using a combination of `alloc` and `init`:

```
Checklist *checklist = [[Checklist alloc] init];
```

But here we're creating a new array using a different method:

```
NSArray *array = [NSArray arrayWithObjects:object1, object2, nil];
```

No `alloc` or `init` in sight. What gives? This form is called a *convenience constructor*. You can actually also write it as:

```
NSArray *array = [[NSArray alloc] initWithObjects:object1, object2, nil];
```

For most intents and purposes these two forms are equivalent. They both allocate and initialize a new `NSArray` object.

Another example that we've been using quite a bit:

```
NSString *string = [NSString stringWithFormat:@"I ate %d ice cream today", 3];
```

This can also be written as:

```
NSString *string = [[NSString alloc] initWithFormat:@"I ate %d ice cream today"↵
    , 3];
```

So why are there two approaches to the same thing? For convenience, mostly. Using `arrayWithObjects` and `stringWithFormat` saves you from typing `alloc`. There is also a historical reason that has to do with memory management, but that went out the door with the arrival of iOS 5.

Because this new view controller is a `UITableViewController`, we have to implement the data source methods for the table view.

» Remove the existing data source stuff from the source file and replace it with:

IconPickerViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
{
  return [icons count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"↵
      IconCell"];

  NSString *icon = [icons objectAtIndex:indexPath.row];
  cell.textLabel.text = icon;
  cell.imageView.image = [UIImage imageNamed:icon];

  return cell;
}
```

Here we get a table view cell and give it text and an image. We will design this cell in the Storyboard editor momentarily. It will be a prototype cell with the cell style "Default"

(or “Basic” as it is called in the Storyboard editor). Cells with this style already contain a text label and an image view, which is very convenient.

» Go to the Storyboard editor. Drag a new Table View Controller from the Object Library next to the List Detail View Controller.

» In the Identity Inspector, change the class of this new table view controller to “IconPickerViewController”.

» Select the prototype cell and set its style to Basic and its (re-use) Identifier to “IconCell”.

That takes care of the design for the icon picker, but we also have to add a new row to the Add/Edit Checklist screen.

» Select the List Detail View Controller and add a static cell. You can do that by dragging a Table View Cell from the Object Library into the table.

» Add a label to this cell and name it “Icon”. System Bold font, size 18. Set the label’s Highlighted color to white.

» Set the cell’s Accessory to Disclosure Indicator.

» Add an Image View to the right of the cell. Make it 36x36 points big. Use the Assistant Editor to add a property for this image view to the `ListDetailViewController` and name it `iconImageView`.

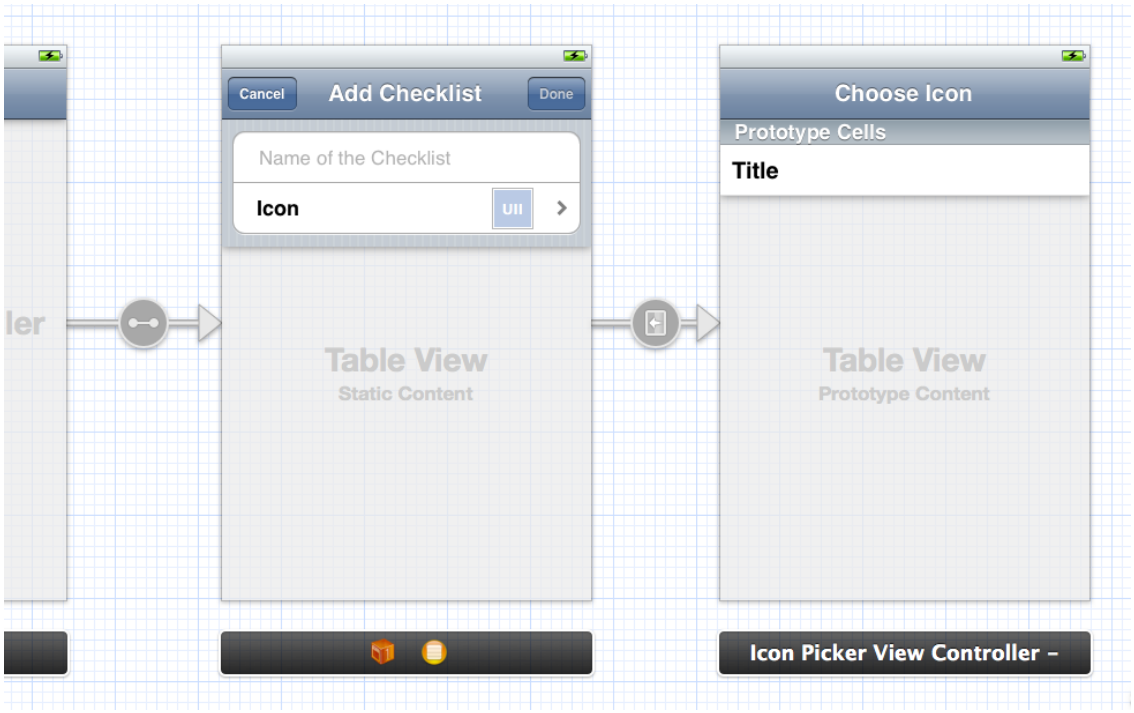
Now that we’ve finished the designs for both screens, we can connect them with a segue.

» Ctrl-drag from the “Icon” table view cell to the Icon Picker View Controller and add a segue. Keep the segue’s style Push but give it an Identifier: “PickIcon”.

» Thanks to the segue, the new view controller has been given a navigation bar. Double-click that navigation bar and change its title to “Choose Icon”.

This part of the Storyboard should now look like this:

The Icon Picker view controller in the Storyboard



» In ListDetailViewController.m, change `willSelectRowAtIndexPath` to:

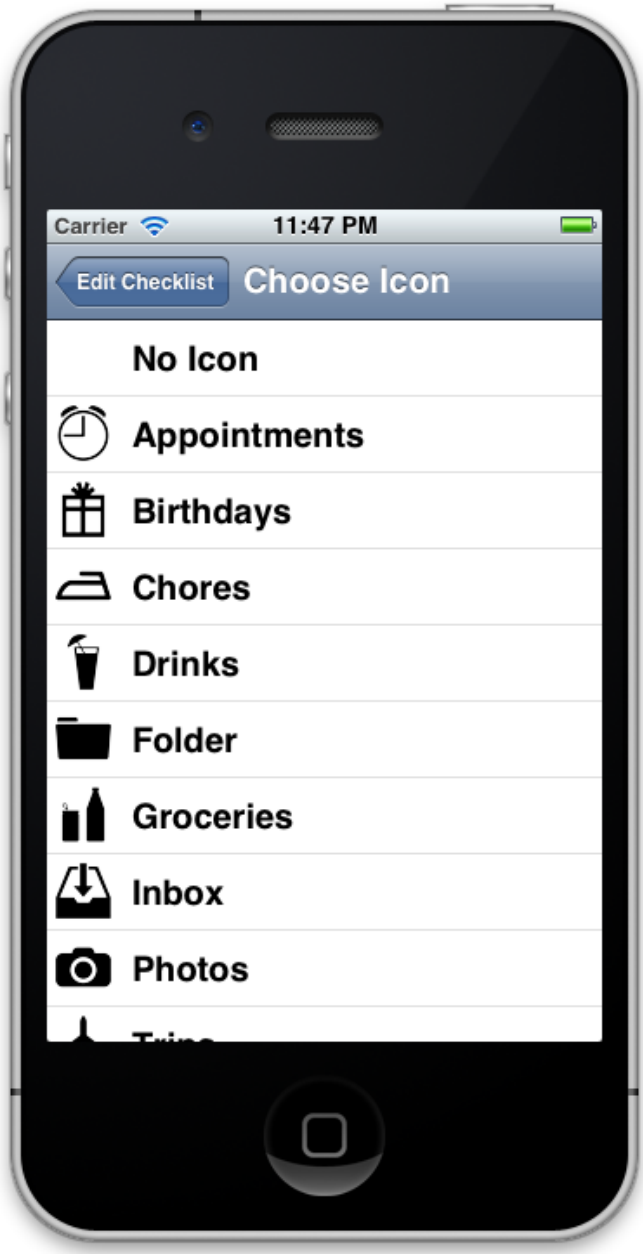
ListDetailViewController.m

```
- (NSIndexPath *)tableView:(UITableView *)tableView willSelectRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  if (indexPath.row == 1) {
    return indexPath;
  } else {
    return nil;
  }
}
```

This is necessary otherwise we cannot tap the row to trigger the segue. Previously this method always returned `nil`, which meant tapping on rows was not possible. Now, however, we want to allow the user to tap on the "Icon" row so we should return the index-path for that row. Users still can't select the first row.

» Run the app and verify that there is now an Icon row in the Add/Edit Checklist screen and that tapping it will open the Choose Icon screen. The icon picker should show a list of icons. You can press the back button to go back.

The icon picker screen



We have to hook up the icon picker to the Add/Edit Checklist screen through its own delegate protocol.

» Add the following to ListDetailViewController.h:

ListDetailViewController.h

```
#import "IconPickerViewController.h"
```

```
. . .

@interface ListDetailViewController : UITableViewController <↵
    UITextFieldDelegate, IconPickerViewControllerDelegate>
```

» Add an instance variable in ListDetailViewController.m:

ListDetailViewController.m

```
@implementation ListDetailViewController {
  NSString *iconName;
}
```

We add an ivar to keep track of the chosen icon name. Even though the `Checklist` object now has an `iconName` property, we cannot keep track of the chosen icon in the `Checklist` object for the simple reason that we may not always have a `Checklist` object, i.e. when we're adding a new checklist. So we'll store the icon name in a temporary variable and copy that into the `Checklist`'s `iconName` property at the right time.

We should initialize the `iconName` variable with something reasonable. Let's go with the folder icon. To do this, we need to add the `initWithCoder` method as this is the method that is used to initialize this view controller (since it's being loaded from a Storyboard).

» Add a new `initWithCoder` method to ListDetailViewController.m:

ListDetailViewController.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    iconName = @"Folder";
  }
  return self;
}
```

This sets the `iconName` variable to `@"Folder"`. This is only necessary for new `Checklists`, which we'll give the Folder icon by default.

» You can get rid of the `initWithStyle` method as it's not being used for anything.

» Update `viewDidLoad` to the following:

ListDetailViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  if (self.checklistToEdit != nil) {
    self.title = @"Edit Checklist";
    self.textField.text = self.checklistToEdit.name;
    self.doneBarButton.enabled = YES;
    iconName = self.checklistToEdit.iconName;
  }

  self.iconImageView.image = [UIImage imageNamed:iconName];
}
```

This has two new lines: If the `checklistToEdit` property is not `nil`, then we copy the `Checklist` object's icon name into our own `iconName` ivar. We also set the icon on the `iconImageView` so it shows up in the Icon row.

We hooked up the Add/Edit Checklist screen to the `IconPickerViewController` with a push segue named "PickIcon". We need to implement `prepareForSegue` in order to tell the `IconPickerViewController` that this screen is now its delegate.

» Add the following method to the bottom of ListDetailViewController.m:

ListDetailViewController.m

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"PickIcon"]) {
    IconPickerViewController *controller = segue.destinationViewController;
    controller.delegate = self;
  }
}
```

Finally, we implement the delegate callback method to remember the name of the chosen icon. We don't call `dismissViewController` here but `popViewControllerAnimated` because the Icon Picker is on the navigation stack, it was not presented modally (we used segue style "Push" instead of "Modal").

» Add to the bottom of ListDetailViewController.m:

ListDetailViewController.m

```
- (void)iconPicker:(IconPickerViewController *)picker didPickIcon:(NSString *)↵
    theIconName
```

```
{
  iconName = theIconName;
  self.iconImageView.image = [UIImage imageNamed:iconName];
  [self.navigationController popViewControllerAnimated:YES];
}
```

This puts the name of the chosen icon into the `iconName` ivar and also updates the image view.

» Change the `done` action to:

ListDetailViewController.m

```
- (IBAction)done
{
  if (self.checklistToEdit == nil) {
    Checklist *checklist = [[Checklist alloc] init];
    checklist.name = self.textField.text;
    checklist.iconName = iconName;

    [self.delegate listDetailViewController:self didFinishAddingChecklist:↵
        checklist];
  } else {
    self.checklistToEdit.name = self.textField.text;
    self.checklistToEdit.iconName = iconName;

    [self.delegate listDetailViewController:self didFinishEditingChecklist:self↵
        .checklistToEdit];
  }
}
```

Here we put the chosen icon name into the `Checklist` object when the user closes the screen.

Finally, we must change `IconPickerViewController` to actually call the delegate method when a row is tapped.

» Add the following method to the bottom of IconPickerViewController.m:

IconPickerViewController.m

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath↵
    *)indexPath
{
  NSString *iconName = [icons objectAtIndex:indexPath.row];
  [self.delegate iconPicker:self didPickIcon:iconName];
}
```

And that's it. You can now set icons on the `Checklist` objects.

We added a new view controller object, designed its user interface in the Storyboard editor, and hooked it up to the Add/Edit Checklist screen using a segue and a delegate. Those are the basic steps you need to take with any new screen that you add.

Making the app look good

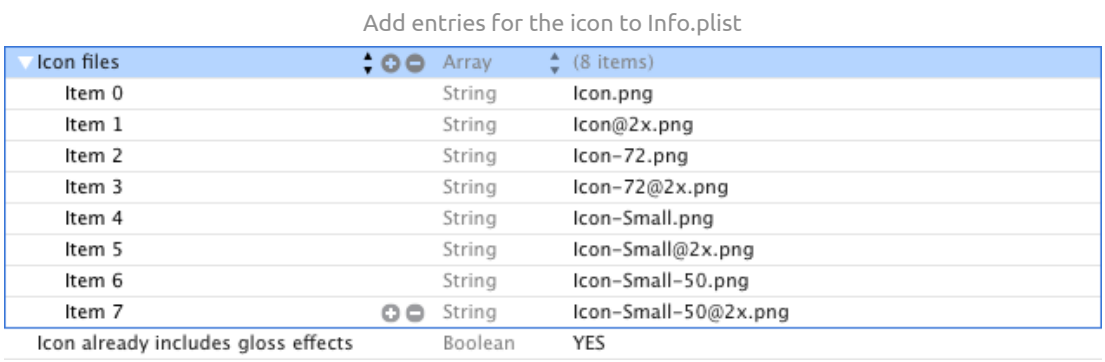
We're going to keep it simple in this tutorial as far as fancying up the graphics goes. The standard look of navigation controllers and table views is perfectly adequate, although a little bland. In the next tutorials you'll see how you can customize the look of these items, especially now that iOS 5 makes this really easy.

For the Checklists app, we'll just add our own icon and a launch image.

The Resources folder for this tutorial contains a folder named "Icon" with the app icon image in various sizes.

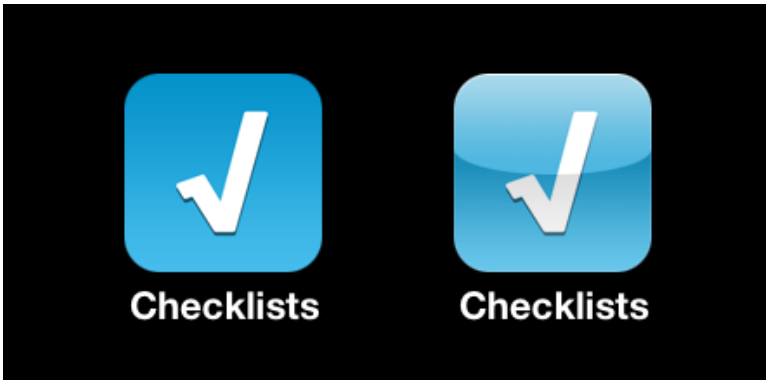
» Add this entire folder to the project.

» In the Project Navigator, locate Checklists-Info.plist and click to open it. Add a new row for "Icon files" and fill in this section as follows:



Also add a row for the setting "Icon already includes gloss effects" and set its value to `YES`. Without this setting, iOS will add a highlight to the icon. That looks good for some icons, but I prefer the icon for this app to have no such gloss.

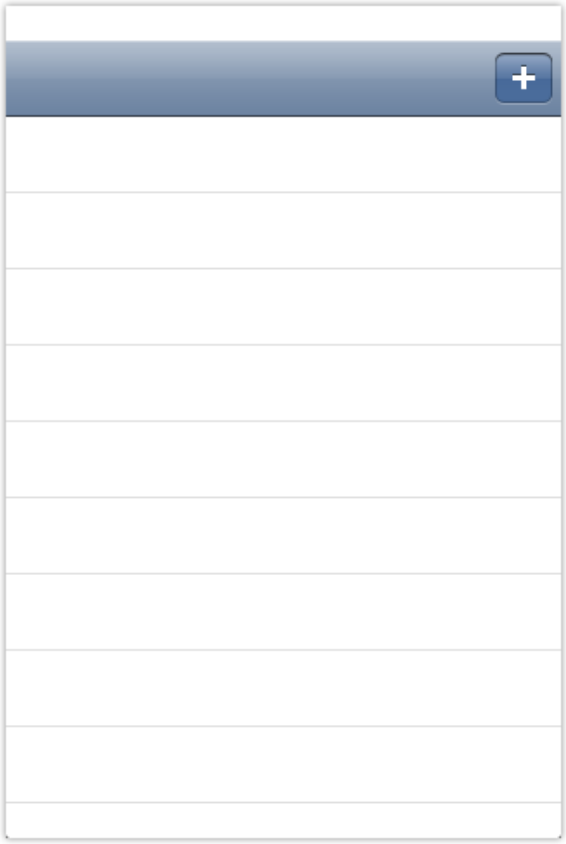
The icon without gloss (left) and with (right)



Besides an icon, apps should also have a launch image. You can find the launch image for this app, in both low-resolution and Retina versions, inside the "Launch Images" folder of this tutorial's Resources. Add that folder to the project.

The launch image simply has a navigation bar without a title but with the Add button, and an empty table view. This will give the illusion the app's UI has already been loaded but that the data hasn't been filled in yet.

The launch image for this app



To make this launch image, I ran the app in the Simulator, then chose File → Save Screen Shot. This puts a new PNG file on your Desktop. Then I opened this image in Photoshop and simply trimmed out the stuff we don't need.

I also blanked out the status bar portion of the image. The iPhone will draw its own status bar on top anyway. You can also cut the status bar off but then Xcode will complain the image is not the recommended size, though in practice that appears to work just as well.

The app also needs a Retina launch image that is twice the size of the regular one, so I repeated this process with the Simulator in Retina mode. (Remember to stop the app in Xcode before you switch the Simulator to Retina mode, or the app will crash hard.)

When you now run the app, instead of a black screen it immediately shows an empty table view with a navigation bar on top. It makes the app look more professional — and faster!

You can find the project files for the app up to this point under "10 - UI Improvements" in the tutorial's Source Code folder.

Bonus Feature: Local Notifications

I hope you're still with me! We have discussed in great detail view controllers, navigation controllers, storyboards, segues, tables and cells, and the data model. These are all essential topics to master if you want to build iOS apps because almost every app uses these building blocks.

In this section we're going to expand the app to add a new feature: local notifications. A *local notification* allows the app to schedule a reminder to the user that will be displayed even when the app is not running. We will add a "due date" field to our `ChecklistItem` object and then remind the user about this deadline with a local notification.

If this sounds like fun, then keep reading. :-)

The steps for this section are as follows:

- Try out a local notification just to see how it works
- Allow the user to pick a due date for to-do items
- Create a date picker control
- Schedule local notifications for the to-do items, and update them when the user changes the due date

Before we'll wonder about how to integrate this in our app, let's just schedule a local notification and see what happens.

By the way, local notifications are different from *push* notifications. Push allows your app to receive messages about external events, such as your favorite team winning the World Series. Local notifications are more similar to an alarm clock: you set a specific time and then it "beeps". Getting push notifications to work is a bit out of the scope of a beginners tutorial, but you can read more about them here [http://www.raywenderlich.com/3443/apple-push-notification-services-tutorial-part-12] if you're interested.

» Open ChecklistsAppDelegate.m and add the following code to the method `didFinishLaunchingWithOptions`:

ChecklistsAppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions↵
    :(NSDictionary *)launchOptions
{
  NSDate *date = [NSDate dateWithTimeIntervalSinceNow:10];

  UILocalNotification *localNotification = [[UILocalNotification alloc] init];
```

```
  localNotification.fireDate = date;
  localNotification.timeZone = [NSTimeZone defaultTimeZone];
  localNotification.alertBody = @"I am a local notification!";
  localNotification.soundName = UILocalNotificationDefaultSoundName;

  [[UIApplication sharedApplication] scheduleLocalNotification:↵
      localNotification];

  return YES;
}
```

The `didFinishLaunchingWithOptions` method is called when the app starts up. We create a new local notification here and tell it to fire 10 seconds after the app has started.

A local notification is scheduled in the future using an `NSDate` object, which specifies a certain date and time. We use the `dateWithTimeIntervalSinceNow` convenience constructor to create an `NSDate` object that points at a time 10 seconds into the future.

We create the `UILocalNotification` object and give it the `NSDate` object as its "fire date". We also set a time zone, so the system automatically adjusts the fire date when the device travels across different time zones (for you frequent flyers).

Local notifications can appear in different ways. We set a text so that an alert message will be shown when the notification fires. We also set a default sound.

Finally, we tell the `UIApplication` object to schedule the notification.

A word on `UIApplication`. We haven't used this object before, but every app has one and it deals with application-wide functionality. You always have to provide a delegate object for `UIApplication` that will handle messages that concern the app as a whole, such as `applicationDidEnterBackground` that we've seen earlier. Our delegate for `UIApplication` is the `ChecklistsAppDelegate` object. The Xcode project templates always provide an app delegate object for you. You won't directly use `UIApplication` a lot, except for special features such as local notifications.

» Add the following method to ChecklistsAppDelegate.m:

ChecklistsAppDelegate.m

```
- (void)application:(UIApplication *)application didReceiveLocalNotification:(↵
    UILocalNotification *)notification
{
  NSLog(@"didReceiveLocalNotification %@", notification);
```

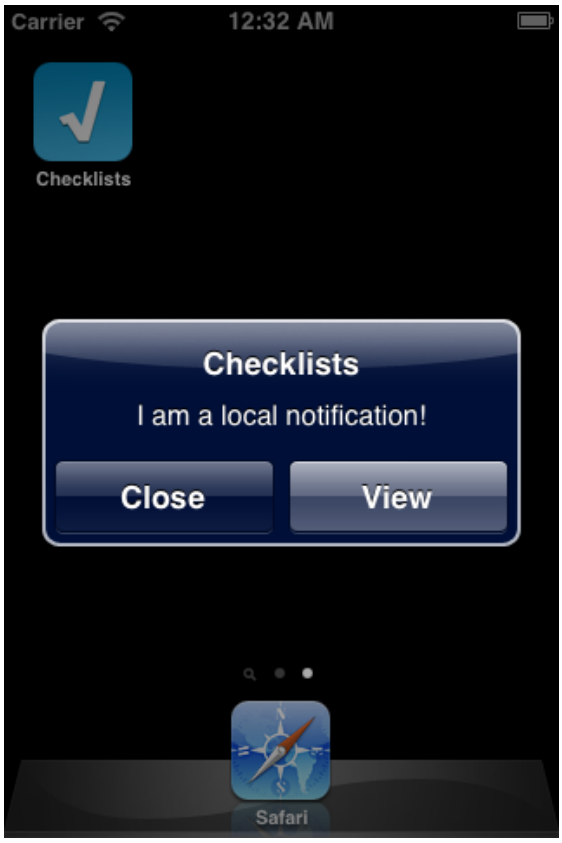
```
}
```

This method will be invoked when the local notification is posted and the app is still running (or suspended). We won't do anything here but for some apps it makes sense to react to it, for example to show whatever thing is the subject of the notification.

» Run the app. Immediately after it has started, press the Home button on the Simulator (or your device if you're running this on your iPhone). Wait 10 seconds.

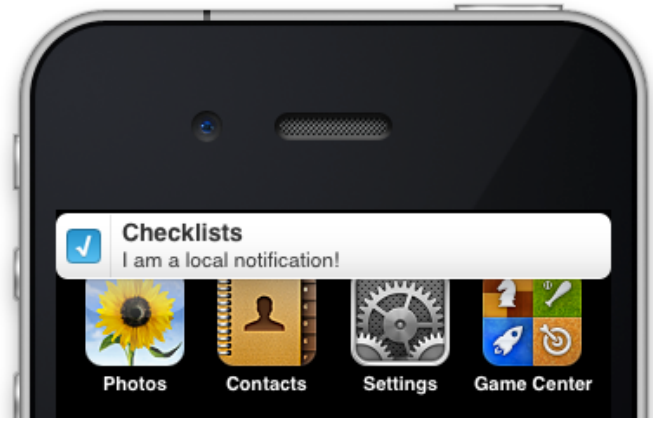
After 10 seconds an alert view should pop up:

The local notification shows up in an alert



The iOS 6 Simulator and your device use Notification Center instead:

The local notification shows up in a Notification Bar



» Press View to go back to the app.

The Debug Area shows that `didReceiveLocalNotification` is called with the notification object. (If you had pressed Close instead, our app would remain in the background and `didReceiveLocalNotification` would not be called.)

The Debug Area should show something like this:

```
Checklists[8022:207] didReceiveLocalNotification <UIConcreteLocalNotification:
0x6e1721O>{fire date = Saturday, July 16, 2011 12:32:38 AM Central European
Summer Time, time zone = Europe/Amsterdam (CEST) offset 72OO (Daylight),
repeat interval = O, repeat count = UILocalNotificationInfiniteRepeatCount,
next fire date = (null)}
```

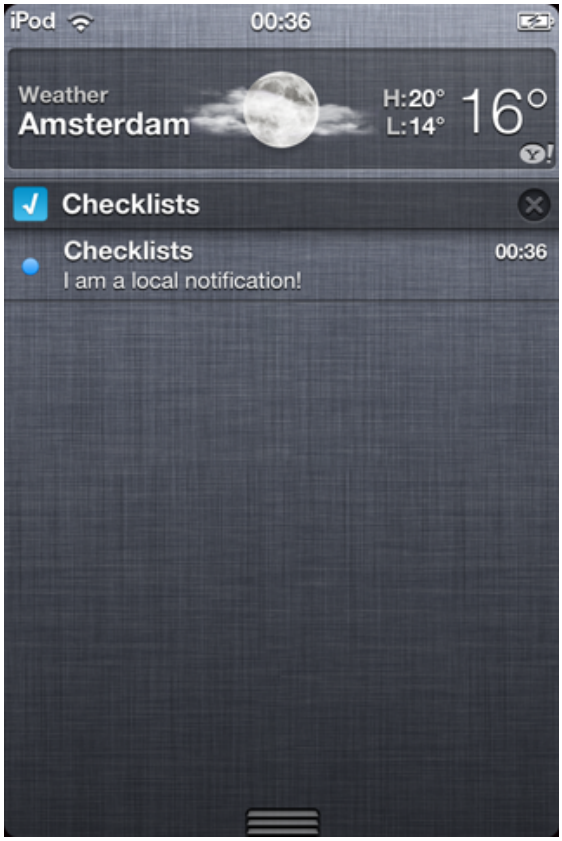
Why did I want you to press the Home button? iOS will only show an alert with the notification message if the app is not currently active.

» Stop the app and run it again. Now don't press Home and just wait.

After 10 seconds you should see the `NSLog` message for `didReceiveLocalNotification` in the Debug Area but no alert is shown. When your app is active, it is supposed to handle any fired notifications in its own manner.

By the way, if you're running this on your device or on the iOS 6 Simulator, you can see the notifications in the Notification Center as well:

The notification in Notification Center



All right, now we know that it works, we should restore `ChecklistsAppDelegate` to its former state because we don't really want to schedule a new notification every time the user starts the app.

» Change the `didFinishLaunchingWithOptions` method back to the way it was:

ChecklistsAppDelegate.m

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions↵
    :(NSDictionary *)launchOptions
{
  return YES;
}
```

You can keep the `didReceiveLocalNotification` method, as it will come in handy when debugging the local notifications.

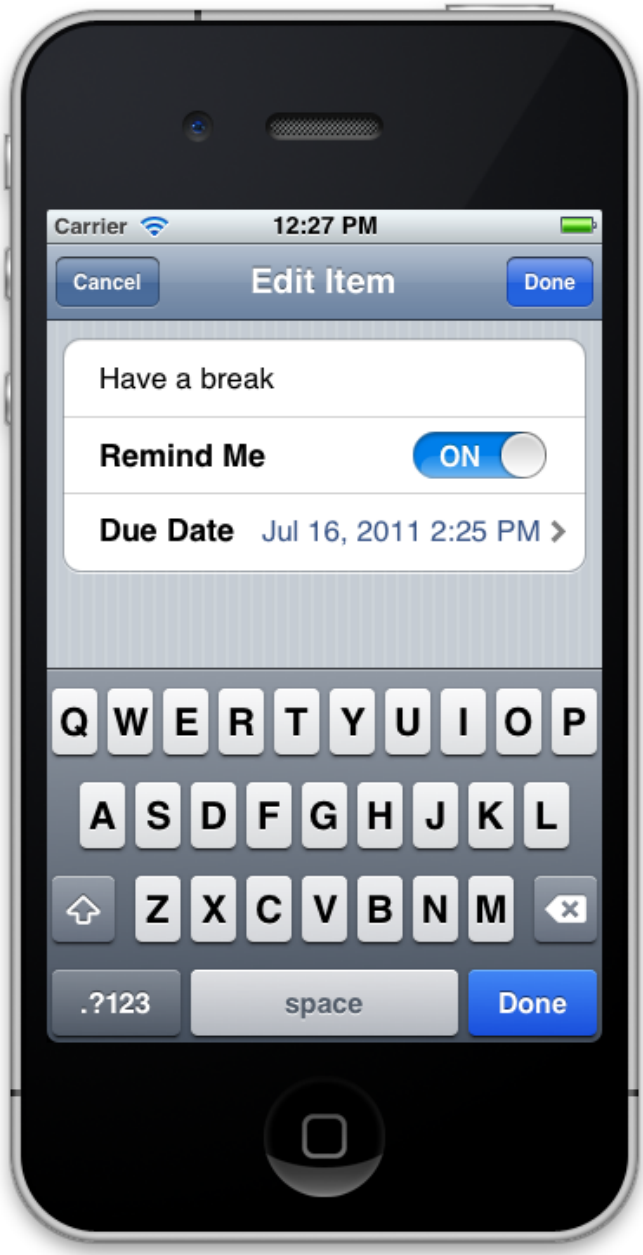
Extending the data model

Let's think about how our app will handle these notifications. Each `ChecklistItem` will get a due date field (an `NSDate` object) and a `BOOL` that says whether we want to be re-

minded of this item or not. You might not want to be reminded of everything, so we shouldn't schedule local notifications for those items. Such a `BOOL` is often called a *flag*. Let's name it `shouldRemind`.

We will add settings for these two new fields to the Add/Edit Item screen and make it look like this:

The Add/Edit Item screen now has Remind Me and Due Date fields



The due date field will require some sort of date picker control. iOS comes with a cool

date picker view but it cannot be used on its own. We'll have to write a view controller for that, probably with its own delegate.

First, let's figure out how and when to schedule the notifications. I can think of the following situations:

- When the user adds a new `ChecklistItem` object that has the `shouldRemind` flag set, we must schedule a new notification.
- When the user changes the due date on an existing `ChecklistItem`, the old notification should be cancelled (if there is one) and a new one scheduled in its place (if `shouldRemind` is still set).
- When the user toggles the `shouldRemind` flag from on to off, the existing notification should be cancelled. The other way around, from off to on, should schedule a new notification.
- When the user deletes a `ChecklistItem`, its notification should be cancelled if it had one.
- When the user deletes an entire `Checklist`, all the notifications for those items should be cancelled.

So we don't need just a way to schedule new notifications but also a way to cancel them. We should probably also check that we don't schedule notifications for to-do items whose due dates are in the past. I'm sure the local notification system can handle that, but let's be good citizens anyway.

`UIApplication` has a method `cancelLocalNotification` that allows us to cancel a notification that was previously scheduled. That method takes a `UILocalNotification` object. Somehow we must associate the `ChecklistItem` object with a `UILocalNotification` in order to be able to cancel that notification.

It is tempting to put the `UILocalNotification` object in `ChecklistItem`, so we always know what it is, but imagine what happens when the app goes to the background. In that case we save the `ChecklistItem` object to the Checklists.plist file — but what about the `UILocalNotification` object? As it happens the `UILocalNotification` conforms to the `NSCoding` protocol so we could serialize it along with the `ChecklistItem` object into our file. However, that is asking for trouble.

These `UILocalNotification` objects are owned by the operating system, not by our app. When the app starts again, it is very well possible that iOS uses different objects to represent the same notifications. We cannot unfreeze these objects from our plist file and expect iOS to recognize them. So let's not store the `UILocalNotification` objects directly.

What will work better is to give the `UILocalNotification` a reference to the associated `ChecklistItem`. Each local notification has an `NSDictionary` named `userInfo` that you can use to store your own values.

We will not use this dictionary to store the `ChecklistItem` object itself, for the same reason as above: when the app closes and later starts again, we will get new `ChecklistItem` objects. Even though they look and behave exactly the same as the old `ChecklistItems` (because we froze and unfroze them), they are likely to be placed elsewhere in memory and the references inside the `UILocalNotifications` will be broken.

Instead of direct references, we will use a numeric identifier. We will give each `ChecklistItem` object a unique numeric ID. Assigning numeric IDs to objects is a common approach when creating data models — it is very similar to giving records in a relational database a numeric primary key, if you're familiar with that sort of thing.

We'll save this ID in the Checklists.plist file and also store it in the `UILocalNotification`'s `userInfo` dictionary. Then we can easily find the notification when we have the `ChecklistItem` object, or the `ChecklistItem` object when we have the notification. This will work even after the app has terminated and all the original objects have long been destroyed.

» Make these changes to ChecklistItem.h:

ChecklistItem.h

```
@property (nonatomic, copy) NSDate *dueDate;
@property (nonatomic, assign) BOOL shouldRemind;
@property (nonatomic, assign) int itemId;
```

» Properties must be synthesized, so add the following in ChecklistItem.m:

ChecklistItem.m

```
@synthesize dueDate, shouldRemind, itemId;
```

Note that I spelled it `itemId`, not `itemID`. That's a stylistic thing; I just like it better that way. I also did not simply call it "id" because `id` is a special keyword in Objective-C (and it has nothing to do with identifiers; for example, we've already seen the `id` keyword when we created our own delegate protocols).

We have to extend `initWithCoder` and `encodeWithCoder` in order to be able to load and save these new properties along with the `ChecklistItem` objects.

» Change these methods:

ChecklistItem.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super init])) {
    self.text = [aDecoder decodeObjectForKey:@"Text"];
    self.checked = [aDecoder decodeBoolForKey:@"Checked"];
    self.dueDate = [aDecoder decodeObjectForKey:@"DueDate"];
    self.shouldRemind = [aDecoder decodeBoolForKey:@"ShouldRemind"];
    self.itemId = [aDecoder decodeIntForKey:@"ItemID"];
  }
  return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
  [aCoder encodeObject:self.text forKey:@"Text"];
  [aCoder encodeBool:self.checked forKey:@"Checked"];
  [aCoder encodeObject:self.dueDate forKey:@"DueDate"];
  [aCoder encodeBool:self.shouldRemind forKey:@"ShouldRemind"];
  [aCoder encodeInt:self.itemId forKey:@"ItemID"];
}
```

That takes care of saving and loading existing objects, but we still have to assign an ID to new objects.

» Replace `ChecklistItem`'s `init` method with the following (or add the method if it doesn't exist yet):

ChecklistItem.m

```
- (id)init
{
  if (self = [super init]) {
    self.itemId = [DataModel nextChecklistItemId];
  }
  return self;
}
```

Because we haven't used the `DataModel` object before in this source file, we have to import it.

» Add the import line at the top of the file:

ChecklistItem.m

```
#import "DataModel.h"
```

Now let's add this new `nextChecklistItemId` method to `DataModel`. As you can guess from its name this method will return a new, unique ID every time you call it.

» Add the method signature to DataModel.h:

DataModel.h

```
+ (int)nextChecklistItemId;
```

» Add the implementation to DataModel.m:

DataModel.m

```
+ (int)nextChecklistItemId
{
  NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
  int itemId = [userDefaults integerForKey:@"ChecklistItemId"];
  [userDefaults setInteger:itemId + 1 forKey:@"ChecklistItemId"];
  [userDefaults synchronize];
  return itemId;
}
```

We're using our old friend `NSUserDefaults` again. This method gets the current "ChecklistItemId" value from `NSUserDefaults`, adds one to it, and writes it back to `NSUserDefaults`. It returns the old value to the caller.

The method also does `[userDefaults synchronize]` to force the `NSUserDefaults` to write these changes to disk immediately, so they won't get lost if we kill the app from Xcode before it had a chance to save. This is important because we don't want two or more `ChecklistItems` to get the same ID.

» Add a default value for "ChecklistItemId" to the `registerDefaults` method:

DataModel.m

```
- (void)registerDefaults
{
  NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt:-1], @"ChecklistIndex",
    [NSNumber numberWithBool:YES], @"FirstTime",
    [NSNumber numberWithInt:0], @"ChecklistItemId",
    nil];

  . . .
}
```

The first time `nextChecklistItemId` is called it will return the ID `0`. The second time it is called it will return the ID `1`, the third time it will return the ID `2`, and so on. The number is incremented by one each time. You can call this method a few billion times before we run out of unique IDs.

Class methods vs instance methods

If you are wondering why we wrote:

```
+ (int)nextChecklistItemId
```

and not:

```
- (int)nextChecklistItemId
```

then I'm glad you're paying attention. :-) Using the + instead of the means that we can call this method without having a reference to the `DataModel` object.

Remember, we did:

```
self.itemId = [DataModel nextChecklistItemId];
```

instead of:

```
self.itemId = [self.dataModel nextChecklistItemId];
```

This is because `ChecklistItem` objects at this moment do not have a reference to the `DataModel` object. We could certainly give them such a reference, but I decided that using a *class method* was easier.

The name of a class method begins with a +. This kind of method applies to the class as a whole. So far we've been using *instance methods*. They begin with a and work only on a specific instance of that class. We haven't discussed the difference between classes and instances before, and we'll get into that in more detail in the next tutorial. For now, just remember that a method starting with a + allows you to call methods on an object even when you don't have a reference to that object.

I had to make a trade-off. Is it worth giving each `ChecklistItem` object a reference to the `DataModel` object, or can I get away with a simple class method? To keep things simple, I chose the latter but it's very well possible that, if we were to develop this app further, it would make more sense to use the references instead.

For a quick test to see if assigning these IDs works, we can put them inside the text that is shown in the `ChecklistItem` cell label. This is just a temporary thing for testing purposes as users couldn't care less about the internal identifier of our objects.

» In ChecklistViewController.m, change the `configureTextForCell` method to:

ChecklistViewController.m

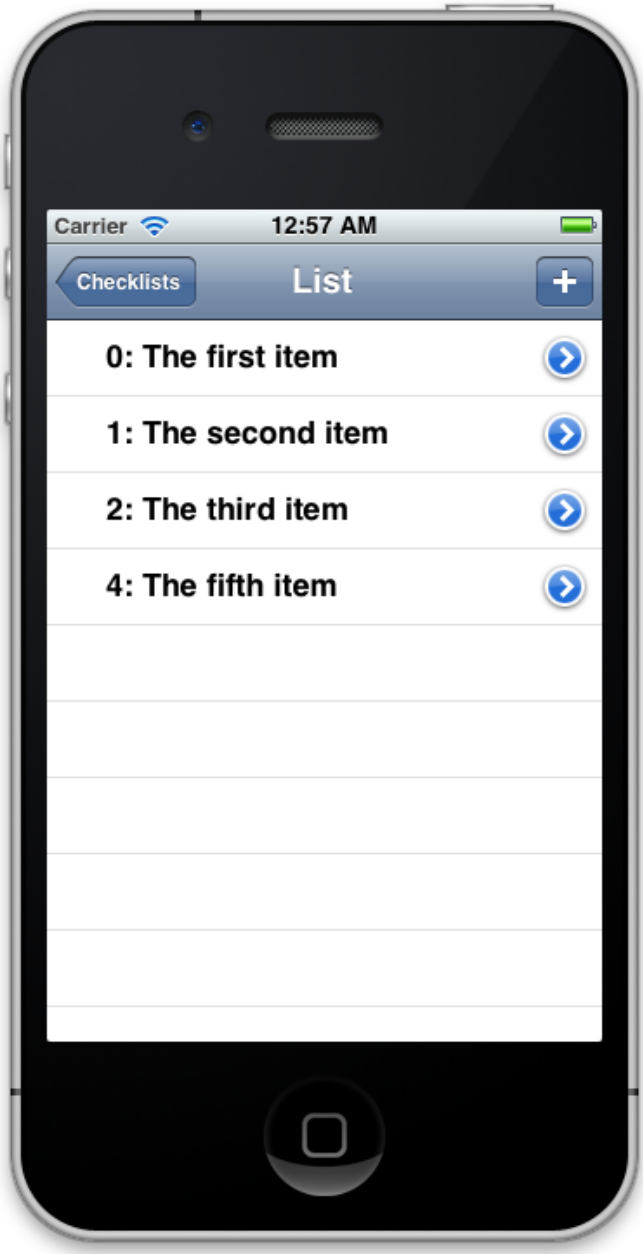
```
- (void)configureTextForCell:(UITableViewCell *)cell withChecklistItem:(↵
    ChecklistItem *)item
{
  UILabel *label = (UILabel *)[cell viewWithTag:1000];
  //label.text = item.text;
  label.text = [NSString stringWithFormat:@"%d: %@", item.itemId, item.text];
}
```

I have commented out the original line because we want to put that back later. The new line uses `stringWithFormat` to add the to-do item's `itemId` property into the text.

» Before you run the app, make sure to delete it from the Simulator first. We have changed the format of the Checklists.plist file again and reading an incompatible file may cause weird crashes.

» Run the app and add some checklist items. Each new item should get a unique identifier. Press Home (to make sure everything is saved properly) and stop the app. Run the app again and add some new items; the IDs for these new items should start counting at where we left off.

The items with their IDs. Note that the item with ID 3 was deleted in this example.



OK, that takes care of the IDs. Now lets add the “due date” and “should remind” fields to the Add/Edit Item screen. (Keep `configureTextForCell` the way it is for the time being; that will come in handy with testing the notifications.)

» Add the following outlets to ItemDetailViewController.h:

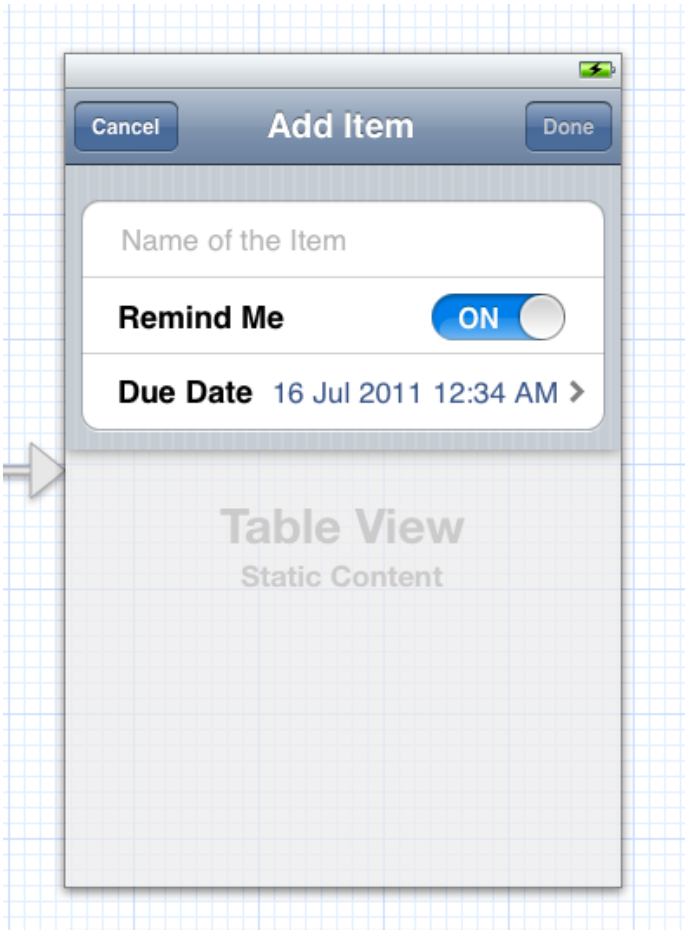
ItemDetailViewController.h

```
@property (nonatomic, strong) IBOutlet UISwitch *switchControl;
@property (nonatomic, strong) IBOutlet UILabel *dueDateLabel;
```

» Go to the Storyboard editor and select the Table View Section in the Item Detail View Controller. In the Attributes Inspector, set the number of rows to 3. This is a quick way to add new static cells. It will duplicate the existing cell including its text field. We don't want text fields in these two new rows, so delete them from the second and third row.

» Design the new cells to look as follows:

The new design of the Add/Edit Item screen



» Add a label to the second cell and give it the text "Remind Me". Font is System Bold, 18. Uncheck Autoshrink and use the Size to Fit command (Cmd =) to make the text fit the label.

» Drag a Switch control into the cell. Hook this switch up to the `switchControl` outlet. (I would have preferred to call this outlet simply "switch", but that is a reserved keyword in the Objective-C language.)

» The third cell has two labels: “Due Date” on the left and the label that will hold the actual chosen date on the right. The Due Date label has a System Bold font, size 18. Set this label’s Highlighted color to white.

» The label on the right should be hooked up to the `dueDateLabel` outlet and is right-aligned. I put a fake date in here just so we know the label is big enough. Make this label as big as possible. For this label, make sure Autoshrink is checked. Font is System, 17. Color is blue (Red: 56, Green: 84, Blue: 135). Set its Highlighted color to white.

» Give the cell a Disclosure Indicator accessory. Set the cell’s Selection attribute to Blue.

If you run the app now and press + to add a checklist item, the app crashes with the following error message:

```
*** Terminating app due to uncaught exception 'NSUnknownKeyException', reason:
'[<ItemDetailViewController 0x6a4bbd0> setValue:forUndefinedKey:]: this class
is not key value coding-compliant for the key dueDateLabel.'
```

I just wanted to show you this because from time to time this happens. We managed to hook up something in Interface Builder or the Storyboard Editor to a property, but when you run the app it turns out this property doesn’t actually exist. Which is correct because we haven’t synthesized `dueDateLabel` yet. Now you know what is going on when you get that kind of error.

Note: The app will not crash if you’re using Xcode 4.4 or better. The latest versions of the Objective-C compiler automatically synthesize properties for you if you do not specify a `@synthesize` statement.

» Add the proper `@synthesize` statements in ItemDetailViewController.m:

ItemDetailViewController.m

```
@synthesize switchControl;
@synthesize dueDateLabel;
```

» Change the `viewDidLoad` method to the following:

ItemDetailViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  if (self.itemToEdit != nil) {
```

```
    self.title = @"Edit Item";
    self.textField.text = self.itemToEdit.text;
    self.doneBarButton.enabled = YES;
    self.switchControl.on = self.itemToEdit.shouldRemind;
    dueDate = self.itemToEdit.dueDate;
  } else {
    self.switchControl.on = NO;
    dueDate = [NSDate date];
  }

  [self updateDueDateLabel];
}
```

If we already have an existing `ChecklistItem` object, we set the switch control on or off, depending on the value of the object's `shouldRemind` property. If we're adding a new item, we always set the switch to off.

For a new item, the due date is right now, `[NSDate date]`. That might not make much sense because by the time you have completed the rest of the fields and pressed Done, that due date will be past. But we do have to suggest something here. An alternative default value could be this time tomorrow, but in most cases the user will have to pick their own due date anyway.

The `updateDueDateLabel` method is new. Add it above `viewDidLoad`:

ItemDetailViewController.m

```
- (void)updateDueDateLabel
{
  NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
  [formatter setDateStyle:NSDateFormatterMediumStyle];
  [formatter setTimeStyle:NSDateFormatterShortStyle];
  self.dueDateLabel.text = [formatter stringFromDate:dueDate];
}
```

To convert the `NSDate` value to text, we use the `NSDateFormatter` object. The way it works is very straightforward, we give it a style for the date component and a separate style for the time component, and then ask it to format our `NSDate` object. You can play with different styles here but space in the label is limited so we can't fit in the full month name, for example.

The cool thing about `NSDateFormatter` is that it takes the current locale into consideration so the time will look good to the user no matter where he is on the globe.

The `dueDate` variable is actually an instance variable, so declare it at the top:

ItemDetailViewController.m

```
@implementation ItemDetailViewController {
  NSDate *dueDate;
}
```

I want to keep track of this date object separately from the text in the `dueDateLabel` because then we don't have to convert back from text in order to get an `NSDate` object.

The last thing to change in this file is the `done` action.

» Change the `done` method to:

ItemDetailViewController.m

```
- (IBAction)done
{
  if (self.itemToEdit == nil) {
    ChecklistItem *item = [[ChecklistItem alloc] init];
    item.text = self.textField.text;
    item.checked = NO;
    item.shouldRemind = self.switchControl.on;
    item.dueDate = dueDate;

    [self.delegate itemDetailViewController:self didFinishAddingItem:item];
  } else {
    self.itemToEdit.text = self.textField.text;
    self.itemToEdit.shouldRemind = self.switchControl.on;
    self.itemToEdit.dueDate = dueDate;

    [self.delegate itemDetailViewController:self didFinishEditingItem:self.↵
        itemToEdit];
  }
}
```

Here we put the value of the switch control and the `dueDate` ivar back into the `ChecklistItem` object when the user presses the Done button.

» Run the app and change the position of the switch control. The app will remember this setting when you terminate it (but be sure to press the Home button first).

The due date row doesn't really do anything yet, however. In order to make that work, we first have to create a date picker.

The date picker

» Add a new `UIViewController` subclass to the project. This time it is a subclass of `UIViewController`, not `UITableViewController`. No nib for user interface. Name it "DatePickerViewController".

» Change the DatePickerViewController.h file to look like this:

DatePickerViewController.h

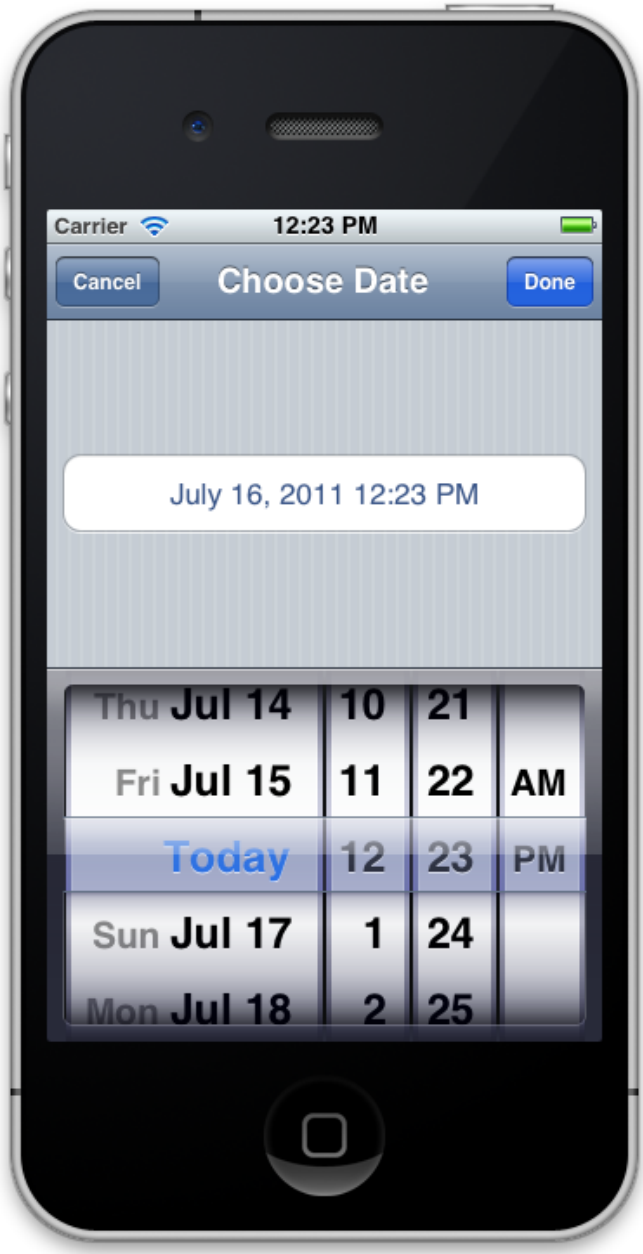
```
#import <UIKit/UIKit.h>

@class DatePickerViewController;

@protocol DatePickerViewControllerDelegate <NSObject>
- (void)datePickerDidCancel:(DatePickerViewController *)picker;
- (void)datePicker:(DatePickerViewController *)picker didPickDate:(NSDate *)↵
    date;
@end

@interface DatePickerViewController : UIViewController <UITableViewDataSource, ↵
    UITableViewDelegate>

@property (nonatomic, strong) IBOutlet UITableView *tableView;
@property (nonatomic, strong) IBOutlet UIDatePicker *datePicker;
@property (nonatomic, weak) id <DatePickerViewControllerDelegate> delegate;
@property (nonatomic, strong) NSDate *date;

- (IBAction)cancel;
- (IBAction)done;
- (IBAction)dateChanged;

@end
```

This should look familiar. We've declared a delegate protocol that the date picker view controller will use to communicate back to the Item Detail View Controller.

The date picker has two outlet properties, `tableView` and `datePicker`, that we'll connect to views inside the Storyboard editor. It also has two regular properties: the `delegate` and a `date`, which is the date we'll initially display on the picker.

The date picker will look like this once it is finished:

The date picker screen



Let's build the UI for the date picker.

» Go to the Storyboard editor and drag a regular View Controller into the canvas, next to the Item Detail View Controller. In the Identity Inspector, set the Class of the new view controller to "DatePickerViewController".

» Ctrl-drag from the Due Date table view cell in the Item Detail View Controller to the new view controller and add a segue. Set the segue style to Modal and give it the identifier

“PickDate”.

That’s right, we’re going to show the date picker as a modal screen on top of the Add/Edit Item screen, which itself is also modal. There’s nothing wrong with that, although you don’t want to stack too many modal screens on top of each other.

In the screenshot of the finished date picker above you can see it has a navigation bar and a Cancel and Done button. We could embed it in a navigation controller as we have done before, but this time let’s fake it.

» From the Object Library drag a Navigation Bar into the Date Picker View Controller. Set its title to “Choose Date”. Drag two Bar Button Items onto the bar and set the left one’s identifier to Cancel and the right one to Done. Use the Connections Inspector to hook up the “selector” from these buttons with their respective actions.

Before we create the rest of this controller, let’s make sure we can open this screen and close it again.

» In DatePickerViewController.m, synthesize the properties:

DatePickerViewController.m

```
@synthesize tableView;
@synthesize datePicker;
@synthesize delegate;
@synthesize date;
```

» Add the `cancel` and `done` actions:

DatePickerViewController.m

```
- (IBAction)cancel
{
  [self.delegate datePickerDidCancel:self];
}

- (IBAction)done
{
  [self.delegate datePicker:self didPickDate:self.date];
}
```

We will make the `ItemDetailViewController` the delegate for this date picker.

» Change the following in ItemDetailViewController.h:

ItemDetailViewController.h

```
#import "DatePickerViewController.h"

. . .

@interface ItemDetailViewController : UITableViewController <↵
    UITextFieldDelegate, DatePickerViewControllerDelegate>
```

» Add the following method to ItemDetailViewController.m:

ItemDetailViewController.m

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
  if ([segue.identifier isEqualToString:@"PickDate"]) {
    DatePickerViewController *controller = segue.destinationViewController;
    controller.delegate = self;
    controller.date = dueDate;
  }
}
```

This sets up the connection with the delegate and we also pass the current date (from the `dueDate` ivar) to the date picker.

What remains is a skeleton implementation of the delegate methods that for now will simply close the date picker screen.

» Add these methods to the bottom of ItemDetailViewController.m:

ItemDetailViewController.m

```
- (void)datePickerDidCancel:(DatePickerViewController *)picker
{
  [self dismissViewControllerAnimated:YES completion:nil];
}

- (void)datePicker:(DatePickerViewController *)picker didPickDate:(NSDate *)↵
    date
{
  [self dismissViewControllerAnimated:YES completion:nil];
}
```

There is one more thing we need to do. When we first created the Add/Edit Item screen, we disabled taps on the rows because that interfered with the text field. We still want to

keep it that way, except for the row with the due date in it. The user should be able to tap that in order to open the date picker. (That's what the disclosure indicator signifies.)

» Change the method `willSelectRowAtIndexPath` to the following, so that taps on the Due Date row are enabled again:

ItemDetailViewController.m

```
- (NSIndexPath *)tableView:(UITableView *)tableView willSelectRowAtIndexPath:(↵
    NSIndexPath *)indexPath
{
  if (indexPath.row == 2) {
    return indexPath;
  } else {
    return nil;
  }
}
```

» Run the app and verify that it works. When you tap on the due date row, the date picker will show up. You should also be able to dismiss the date picker with Cancel and Done.

Let's finish the design of the date picker screen.

» Take a Date Picker wheel from the Object Library and drop it into the view controller. Connect it to the `datePicker` outlet. Also connect its Value Changed event to the `dateChanged` action.

» In the DatePickerViewController.m, implement these methods:

DatePickerViewController.m

```
- (void)viewWillAppear:(BOOL)animated
{
  [super viewWillAppear:animated];

  [self.datePicker setDate:self.date animated:YES];
}

- (IBAction)dateChanged
{
  self.date = [self.datePicker date];
}
```

In `viewWillAppear` we set the current date that will be shown in the `UIDatePicker` view to the value from the `date` property. In `dateChanged` we do the opposite and we store the date picker's chosen date back into the property.

» In ItemDetailViewController.m, change the date picker delegate method to:

ItemDetailViewController.m

```
- (void)datePicker:(DatePickerViewController *)picker didPickDate:(NSDate *)↵
    date
{
  dueDate = date;
  [self updateDueDateLabel];

  [self dismissViewControllerAnimated:YES completion:nil];
}
```

We put the date picker's date into our own `dueDate` ivar and update the corresponding label.

» Try it out. Run the app and pick a due date. It should now be remembered along with the `ChecklistItem` even when you quit the app.

(If the app crashes at this point, then remove it from the Simulator and try again. Because the format of the Checklists.plist file has changed a while back, the `dueDate` field may be read as `nil` and the date picker doesn't like it when we give it a `nil NSDate` object in the `viewWillAppear` method. This won't happen with a Checklists.plist file that is up-to-date but it can occur if you're still using an older version.)

The date picker view controller still looks a little bare, so let's return to the Storyboard editor and dress it up.

» Drag a Table View into the white part of the view controller. You'll have to resize the table view so that it only takes up the top part of the screen (make it 200 points high). We'll be using this table view to show the picked date, it doesn't have any other function at all.

» Connect the table view to the `tableView` outlet. Connect the `dataSource` and `delegate` outlets to the view controller. Set the table view style to Grouped.

This would be an ideal situation for using static cells, we only need one cell with a label. One problem: static cells only work if the view controller is a `UITableViewController` and ours isn't. We can't really make it one either because a table view controller assumes that it has just one view: the table view. But this screen has a date picker view as well, so static cells are out the window and we'll just have to make do with a prototype cell.

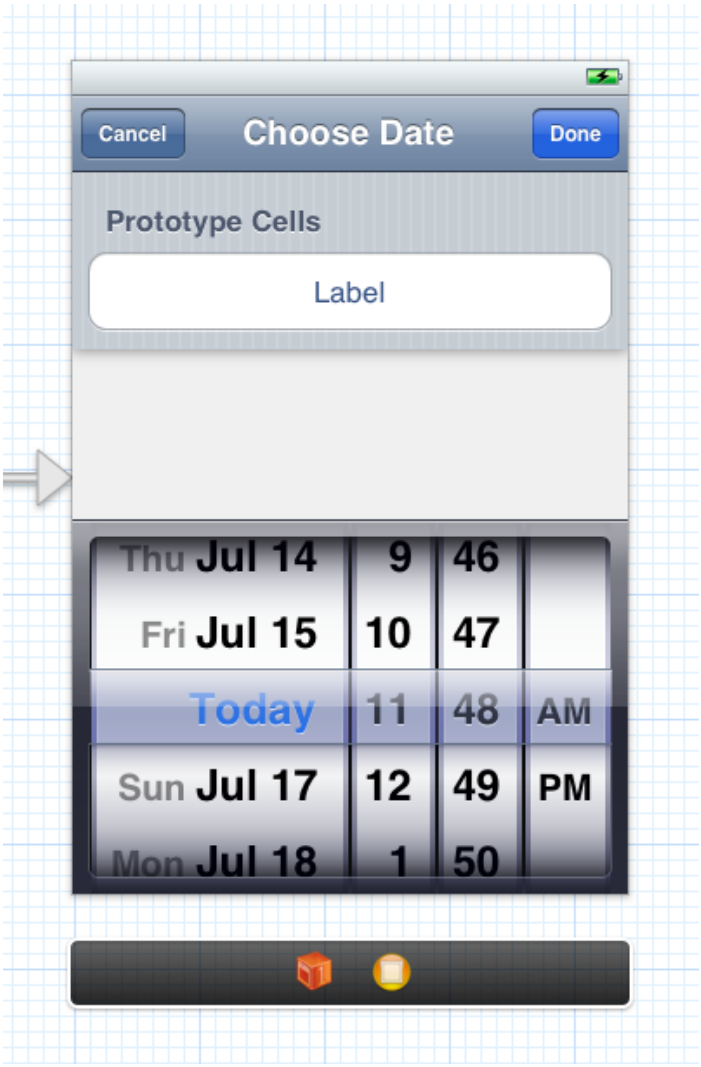
» Drag a label into the prototype cell and make it stretch the entire width. Set the following attributes. Alignment: Center. Font: System, size 17. Text Color: the dark blue we used before (pick it from the Recently Used Colors list). Tag: 1000.

» Select the table view cell. Set its Reuse Identifier to “DateCell”. Set the cell’s Selection property to None.

» Select the table view. Uncheck the Scrolling Enabled attribute. Now the user can no longer “bounce” the table view. It will just appear in a fixed position.

The final design of the Date Picker View Controller should look like this:

The design of the date picker screen



» In DatePickerViewController.m, add the following methods:

DatePickerViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(↵
    NSInteger)section
{
```

```
  return 1;
}

- (UITableViewCell *)tableView:(UITableView *)theTableView ↵
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
  UITableViewCell *cell = [theTableView dequeueReusableCellWithIdentifier:@"↵
      DateCell"];

  dateLabel = (UILabel *)[cell viewWithTag:1000];
  [self updateDateLabel];

  return cell;
}

- (NSIndexPath *)tableView:(UITableView *)theTableView willSelectRowAtIndexPath↵
    :(NSIndexPath *)indexPath
{
  return nil;
}
```

These are the data source and delegate methods for the table view. Any taps on the row will be ignored.

The `cellForRowAtIndexPath` method does something special, it looks up the label with tag 1000 and then puts it into the `dateLabel` ivar.

» Add this ivar to the `@implementation` section:

DatePickerViewController.m

```
@implementation DatePickerViewController {
  UILabel *dateLabel;
}
```

We're not storing `dateLabel` in a property because it's not really an outlet. We're just using it for convenience so that we can refer to this label directly from now on without having to find the table view cell first and doing `[viewWithTag:1000]` again.

» Above `dateChanged`, add the following method:

DatePickerViewController.m

```
- (void)updateDateLabel
{
  NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
  [formatter setDateStyle:NSDateFormatterLongStyle];
```

```
  [formatter setTimeStyle:NSDateFormatterShortStyle];
  dateLabel.text = [formatter stringFromDate:self.date];
}
```

This is similar to the date formatting we did in `ItemDetailViewController`, except this time we use the long style for the date.

» Update the `dateChanged` action to call this method:

DatePickerViewController.m

```
- (IBAction)dateChanged
{
  self.date = [self.datePicker date];
  [self updateDateLabel];
}
```

If you run the app now, everything should work. Except for a small detail: the label is positioned at the top of the screen and it would be nicer to have it vertically centered in the table view.

» Add the following method to DatePickerViewController.m:

DatePickerViewController.m

```
- (CGFloat)tableView:(UITableView *)tableView heightForHeaderInSection:(↵
    NSInteger)section
{
  return 77;
}
```

This is another method from the table view delegate protocol. Each section of a table can have a header. We only have one section and the header is empty, but with this method we can determine how tall that header is. In effect, this lets us add 77 points of empty space above our one and only row, centering it inside the table view.

And that should do it.

Note that you can take this date picker and drop it into any other project without too much trouble. It is not tied to the data model of this particular app. It doesn't depend on `ChecklistItem` objects, only on `NSDate`. That is a good design.

You should always be thinking: can I turn this object into something that I can easily re-use in other

projects? Picking dates is a common thing so it's good to have a component like this in your library. You only have to write it once and then you can plug it into all your future projects.

Scheduling the local notifications

One of the principles of Object-Oriented Programming is that objects can do as much as possible themselves. Therefore, it makes sense that the `ChecklistItem` object can schedule its own notifications.

» Add the following method declaration to ChecklistItem.h:

ChecklistItem.h

```
- (void)scheduleNotification;
```

» Add the method itself to ChecklistItem.m:

ChecklistItem.m

```
- (void)scheduleNotification
{
  if (self.shouldRemind && [self.dueDate compare:[NSDate date]] != ↵
      NSOrderedAscending) {
    NSLog(@"We should schedule a notification");
  }
}
```

This compares the due date on the item with the current date. If the due date is in the past, then the `NSLog()` will not be performed. Note the use of the `&&` "and" operator. We only print the text when the Remind Me switch is set to "on" *and* the due date is in the future.

We will call this method when the user presses the Done button after adding or editing a to-do item.

» Change the `done` action in ItemDetailViewController.m:

ItemDetailViewController.m

```
- (IBAction)done
{
  if (self.itemToEdit == nil) {
    ChecklistItem *item = [[ChecklistItem alloc] init];
    item.text = self.textField.text;
    item.checked = NO;
```

```
    item.shouldRemind = self.switchControl.on;
    item.dueDate = dueDate;
    [item scheduleNotification];

    [self.delegate itemDetailViewController:self didFinishAddingItem:item];
  } else {
    self.itemToEdit.text = self.textField.text;
    self.itemToEdit.shouldRemind = self.switchControl.on;
    self.itemToEdit.dueDate = dueDate;
    [self.itemToEdit scheduleNotification];

    [self.delegate itemDetailViewController:self didFinishEditingItem:self.↵
        itemToEdit];
  }
}
```

Only the lines with `[scheduleNotification]` are new.

» Run the app and try it out. Add a new item, set the switch to ON but don't change the due date. Press Done. There should be no message in the Debug Area because the due date has already passed.

» Add another item, set the switch to ON, and choose a due date in the future. When you press Done now, there should be an `NSLog` ("We should schedule a notification") in the Debug Area.

Now that we've verified the method is called in the proper place, let's actually schedule a new `UILocalNotification` object. We will first consider the case of a new to-do item being added.

» In ChecklistItem.m, change `scheduleNotification` to:

ChecklistItem.m

```
- (void)scheduleNotification
{
  if (self.shouldRemind && [self.dueDate compare:[NSDate date]] != ↵
      NSOrderedAscending) {

    UILocalNotification *localNotification = [[UILocalNotification alloc] init↵
        ];
    localNotification.fireDate = self.dueDate;
    localNotification.timeZone = [NSTimeZone defaultTimeZone];
    localNotification.alertBody = self.text;
    localNotification.soundName = UILocalNotificationDefaultSoundName;
    localNotification.userInfo = [NSDictionary dictionaryWithObject:[NSNumber ↵
        numberWithInt:self.itemId] forKey:@"ItemID"];
```

```
    [[UIApplication sharedApplication] scheduleLocalNotification:↵
        localNotification];

    NSLog(@"Scheduled notification %@ for itemId %d", localNotification, self.↵
        itemId);
  }
}
```

You've seen this code before. We create a `UILocalNotification` object. This time, however, we use the `ChecklistItem`'s `dueDate` and `text`. We also add a `userInfo` dictionary with the item's ID as the only contents. That is how we'll be able to find this notification later in case we need to cancel it.

» Test it out. Run the app, add a new checklist item, set the due date a minute into the future, press Done and tap the Home button on the Simulator. Now wait one minute and the notification should appear. Pretty cool!

The date picker doesn't show you seconds but they still are there (just watch the `NSLog` output). If you set the due date to 10:16 PM when it's currently 10:15:54 PM, you'll have to wait until exactly 10:16:54 for the event to fire. It would probably be a better user experience if you always set the seconds to 0, but that's a topic for another day.

That takes care of the case where we're adding a new notification. There are two situations left: the user edits an existing item and the user deletes an item. Let's do editing first.

When the user edits an item, the following situations can occur:

- Remind Me was switched off and is now switched on. We have to schedule a new notification.
- Remind Me was switched on and is now switched off. We have to cancel the existing notification.
- Remind Me stays switched on but the due date changes. We have to cancel the existing notification and schedule a new one.
- Remind Me stays switched on but the due date doesn't change. We don't have to do anything.
- Remind Me stays switched off. Here we also don't have to do anything.

Of course, in all those situations we'll only schedule the notification if the due date is in the future.

Phew, that's quite a list. It's always a good idea to take stock of all possible scenarios before you start programming because this gives you a clear picture of everything you need to tackle.

It may seem like we need to write a lot of logic here to deal with all these situations, but actually it turns out to be quite simple. First we'll look if there is an existing notification for this to-do item. If there is, we simply cancel it. Then we determine whether the item should have a notification and if so, we schedule a new one. That should take care of all the above situations, even if sometimes we simply could have left the existing notification alone. Crude, but effective.

» Add the following to the top of `scheduleNotification`:

ChecklistItem.m

```
- (void)scheduleNotification
{
  UILocalNotification *existingNotification = [self notificationForThisItem];
  if (existingNotification != nil) {
    NSLog(@"Found an existing notification %@", existingNotification);
    [[UIApplication sharedApplication] cancelLocalNotification:↵
        existingNotification];
  }

  . . .
}
```

This calls a method `notificationForThisItem`, which we'll add in a second. If that method returns a valid `UILocalNotification` object (i.e. not `nil`), then we dump some debug info using `NSLog()` and then ask the `UIApplication` object to cancel this notification.

» Add the new `notificationForThisItem` method above `scheduleNotification`:

ChecklistItem.m

```
- (UILocalNotification *)notificationForThisItem
{
  NSArray *allNotifications = [[UIApplication sharedApplication] ↵
      scheduledLocalNotifications];
  for (UILocalNotification *notification in allNotifications) {
    NSNumber *number = [notification.userInfo objectForKey:@"ItemID"];
    if (number != nil && [number intValue] == self.itemId) {
      return notification;
    }
  }
  return nil;
```

```
}
```

This asks `UIApplication` for a list of all scheduled notifications. Then we loop through that list and look at each notification one-by-one. It should have an "ItemID" value inside the `userInfo` dictionary. If that value exists and equals our item ID, then we've found a notification that belongs to this `ChecklistItem`. If none of the local notifications match, or there aren't any to begin with, the method returns `nil`.

This is a common pattern that you'll see in a lot of code. Something returns an array of items and you loop through the array to find the first item that matches what you're looking for, in this case the item ID. Once you've found it, you can exit the loop.

» Run the app and try it out. Add a to-do item with a due date a few days into the future. A new notification will be scheduled. Edit the item and change the due date. The old notification will be removed and a new one scheduled for the new date.

» Edit the to-do item again but now set the switch to OFF. The old notification will be removed and no new notification will be scheduled. Edit again and don't change anything; no new notification will be scheduled because the switch is still off. This should also work if you terminate the app in between.

One last case to handle: deletion of the `ChecklistItem` object. This can happen in two ways: 1) the user can delete an individual item using swipe-to-delete; 2) the user can delete an entire checklist in which case all its `ChecklistItem` objects are also deleted.

An object is notified when it is about to be deleted using the `dealloc` message. We can simply implement this method, look if there is a scheduled notification for this item and then cancel it.

» Add the following to the bottom of ChecklistItem.m:

ChecklistItem.m

```
- (void)dealloc
{
  UILocalNotification *existingNotification = [self notificationForThisItem];
  if (existingNotification != nil) {
    NSLog(@"Removing existing notification %@", existingNotification);
    [[UIApplication sharedApplication] cancelLocalNotification:↵
        existingNotification];
  }
}
```

That's all we have to do. The `dealloc` method will be invoked when we delete an individual `ChecklistItem` but also when we delete a whole `Checklist` (because then all its `ChecklistItems` will be destroyed as well, as the array they are in is deallocated).

» Run the app and try it out. First schedule some notifications far into the future (so they won't be fired when you're testing) and then remove that to-do item or its entire checklist. You should now see a message in the Debug Area.

Once you're convinced everything works, you can remove the `NSLog()` statements. They are only temporary for debugging purposes. You probably don't want to leave them in the final app. (They won't hurt any, but the end user can't see those messages anyway.)

» You should also remove the item ID from the label in the `ChecklistViewController`, we no longer need that.

You can find the project files for the app up to this point under "11 - Local Notifications" in the tutorial's Source Code folder.

Exercise: Put the due date in a label on the table view cells under the text of the to-do item. ■

Exercise: Sort the to-do items list based on the due date. This is similar to what we did with the list of `Checklists` except that now you're sorting `ChecklistItem` objects and you'll be comparing `NSDate` objects instead of `NSString`s. (`NSDate` does not have a `localizedStandardCompare` method but it does have a regular `compare`). ■

One last bug...

The app may seem to work perfectly now, but I'm afraid there is still a bug if you run it on iOS 5 — but not on iOS 6.0 or higher. This bug is not something that will make the app crash, but it could potentially upset or confuse users. Here's how to reproduce it:

» Run the app. Add a new to-do item. Type something for the name of the item. Set Remind Me to ON. Tap on the Due Date row to open the date picker. From the Simulator's menubar choose Hardware → Simulate Memory Warning.

The Debug Area will now say something like:

```
Checklists[1417:207] Received memory warning.
```

» Press Cancel to close the date picker screen. When you return to the Add Item screen, the text you previously typed is gone and Remind Me is back to the OFF state.

What happened here? When your app gets a low memory warning, UIKit will unload the views of any controllers that are not currently visible. When you're interacting with the date picker screen and a low memory warning comes in, UIKit will unload the views of the Add/Edit Item screen, the Checklist screen and the All Lists screen. That is, it throws away the table view, the text field, the switch control, and their contents.

Note: UIKit only unloads views in low memory situations on iOS 5 but it no longer does so on iOS 6. Apple decided that this mechanism was too confusing and they simplified how UIKit deals with memory running low. That's why this bug doesn't appear on iOS 6.0 or later. Still, if you want your apps to be compatible with iOS 5, it's worth handling these sorts of situations.

In a previous version of the iOS Apprentice, I always recommended that you set your outlet properties to `nil` in the `viewDidUnload` method. If you don't set your outlet properties to `nil`, then not all memory can be properly released when the view is unloaded. UIKit is pretty smart about releasing memory but it can't know what to do with outlet properties that you have created. As long as you're keeping a reference to a `UITextField`, for example, that text field object stays in memory. Only when all references to it are dropped can the text field be deallocated.

To make sure UIKit is not longer keeping references to these objects, we're responsible for setting our outlet properties to `nil`. That used to be the purpose of `viewDidUnload`, it gave us a chance to clean up. However, Apple has deprecated this method and as of iOS 6 it will no longer be called. Because we still want the Checklists app to do the right thing on iOS 5, I will show you the new recommended way for catching low memory warnings and dealing with them.

» Add the following method to ItemDetailViewController.m:

ItemDetailViewController.m

```
- (void)didReceiveMemoryWarning
{
  [super didReceiveMemoryWarning];

  if ([self isViewLoaded] && self.view.window == nil) {
    self.view = nil;
  }

  if (![self isViewLoaded]) {
    self.textField = nil;
    self.doneBarButton = nil;
    self.switchControl = nil;
    self.dueDateLabel = nil;
  }
}
```

The `didReceiveMemoryWarning` method is called when memory gets low. You can override it to free up any memory you no longer need. By the time we get here on iOS 5, the view has already been unloaded and we just need to set the outlet properties to `nil` as well. However, on iOS 6 we first force the view to unload itself by setting `self.view` to `nil`, so that the app works the same across all versions of iOS. Try it out on the iOS 6 Simulator; after simulating a low memory warning, the text field is empty here too.

When you press Cancel to go back to the Add/Edit Item screen, UIKit will reload the view for that screen and resets it to its default state. It also calls `viewDidLoad` again, which either empties out the contents of the text field and sets the switch to "off" (when adding a new item) or overwrites the contents of the text field and switch control with the values from the `ChecklistItem` object (when editing an existing item). Either way, whatever we typed into the text field is gone.

Loading data into your controls inside `viewDidLoad` works fine if you never open a new view controller on top of this one because then UIKit will never unload the view in a low-memory situation (it would be strange if the current screen simply disappeared). But now that we open the date picker screen, our view can be unloaded at any time and there is nothing we can do (or should do!) to stop it.

We need to make the `ItemDetailViewController` a little smarter so that it doesn't throw away the changes the user has made when this happens.

» Change `viewDidLoad` to the following:

ItemDetailViewController.m

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  if (self.itemToEdit != nil) {
    self.title = @"Edit Item";
  }

  self.textField.text = text;
  self.switchControl.on = shouldRemind;

  [self updateDoneBarButton];
  [self updateDueDateLabel];
}
```

From now on, we will store the name of the `ChecklistItem` in the new `text` ivar and the state of the switch control in `shouldRemind`. (We already kept a copy of the date in the `dueDate` ivar.) When the view is loaded, we put those values into the UI controls.

The idea is that when the view gets unloaded out from under us, we still have the proper values in the `text`, `shouldRemind` and `dueDate` ivars. When the view gets restored again, we put those values back in the text field and the switch control and it will seem like nothing ever happened.

» The `updateDoneBarButton` method is new. Add it above `viewDidLoad`:

ItemDetailViewController.m

```
- (void)updateDoneBarButton
{
  self.doneBarButton.enabled = ([text length] > 0);
}
```

» Change the text field delegate method to:

ItemDetailViewController.m

```
- (BOOL)textField:(UITextField *)theTextField shouldChangeCharactersInRange:(↵
    NSRange)range replacementString:(NSString *)string
{
  text = [theTextField.text stringByReplacingCharactersInRange:range withString↵
      :string];
  [self updateDoneBarButton];
  return YES;
}
```

This now copies the contents of the text field into the `text` ivar and automatically enables or disables the Done button if necessary. The `text` ivar will always have a backup copy of what is in the text field, so we will remember those contents when the view gets unloaded.

» Add the following method:

ItemDetailViewController.m

```
- (void)textFieldDidEndEditing:(UITextField *)theTextField
{
  text = theTextField.text;
  [self updateDoneBarButton];
}
```

This is also a text field delegate method. We need it for the following situation: If there is a spelling suggestion and you press Done on the keyboard, then the text in the text field does change but we do not get a `shouldChangeCharactersInRange` notification for some reason. So we handle that situation in `textFieldDidEndEditing`.

» Add the following method declaration to ItemDetailViewController.h:

ItemDetailViewController.h

```
- (IBAction)switchChanged:(UISwitch *)sender;
```

» In the Storyboard editor, hook up the switch control's Value Changed event to this action method.

» Add the method implementation to ItemDetailViewController.m:

ItemDetailViewController.m

```
- (IBAction)switchChanged:(UISwitch *)sender
{
  shouldRemind = sender.on;
}
```

Now the contents of the `text` and `shouldRemind` ivars always mirror what is visible on the screen. We still need to add the actual ivars and give them initial values.

» Add the instance variables:

ItemDetailViewController.m

```
@implementation ItemDetailViewController {
  NSString *text;
  BOOL shouldRemind;
  NSDate *dueDate;
}
```

» Add the `initWithCoder` method:

ItemDetailViewController.m

```
- (id)initWithCoder:(NSCoder *)aDecoder
{
  if ((self = [super initWithCoder:aDecoder])) {
    text = @"";
    shouldRemind = NO;
    dueDate = [NSDate date];
  }
  return self;
}
```

» You can remove `initWithStyle` as that method is never used.

» Run the app and add a new to-do item. Type something in the Name field, toggle the Remind Me switch to ON, and click Due Date. When the date picker appears, choose Simulate Memory Warning from the Simulator's Hardware menu. Now press Cancel to close the date picker. Lo and behold: this time the Add Item screen did remember the text and the state of the Remind Me switch.

There is one more thing we need to do. When you tap an existing item to edit it, the text field is empty, the Remind Me switch is always OFF, and the Due Date is always today's date and the current time. That ain't right! Due to the recent changes in `viewDidLoad`, we never load the `ChecklistItem`'s values into the UI controls anymore.

We're going to use a little trick to solve this problem.

» Add the following method to ItemDetailViewController.m:

ItemDetailViewController.m

```
- (void)setItemToEdit:(ChecklistItem *)newItem
{
  if (itemToEdit != newItem) {
    itemToEdit = newItem;
    text = itemToEdit.text;
    shouldRemind = itemToEdit.shouldRemind;
    dueDate = itemToEdit.dueDate;
  }
}
```

When this method is called, it takes a `ChecklistItem` object and puts it into the `itemToEdit` ivar. It also copies the item's `text`, `shouldRemind` and `dueDate` properties in the temporary ivars that we have recently introduced.

So what is the trick? The `setItemToEdit` method is a so-called *setter*. This method is invoked automatically when you do:

```
controller.itemToEdit = anItem;
```

That is exactly what we do in `ChecklistViewController` when we perform the segue to `ItemDetailViewController` (in the `prepareForSegue` method). When you set a property to a new value, the corresponding setter method is invoked.

Normally you don't have to write setter methods, as that is what `@synthesize` provides for you. In this case, however, we don't just want to put a new value into the `itemToEdit`

property, we also want to fill in the `ItemDetailViewController`'s `text`, `shouldRemind` and `dueDate` variables. By making our own setter, we can do that conveniently in one place, and before `viewDidLoad` happens.

» Run the app and verify that everything now works again.

If the iPhone runs out of available memory while the date picker is showing, then the `ItemDetailViewController`'s view may get unloaded but at least we keep track of the data that the user entered. In other words, we've taken the responsibility for remembering the data out of the views (the text field and switch control) and used a mini-data model to remember them (the new instance variables).

This is how I like to write all my modal view controllers: put all the data into separate ivars and update these ivars when the user changes something in the UI.

Exercise: `ListDetailViewController` has a similar problem if you're inside the Icon Picker and a low memory warning occurs. I'm sure you can fix that issue now you know how. ■

You can find the final project files for the Checklists app under "12 - Low Memory" in the tutorial's Source Code folder.

I've pointed out a few times already that users of Xcode 4.4 or better no longer need to type `@synthesize` for their properties. But if you've been skipping the synthesize statements, then the compiler now gives a bunch of errors on the new `setItemToEdit:` method.

To fix this, change the code to:

```
- (void)setItemToEdit:(ChecklistItem *)newItem
{
    if (_itemToEdit != newItem) {
        _itemToEdit = newItem;
        text = _itemToEdit.text;
        shouldRemind = _itemToEdit.shouldRemind;
        dueDate = _itemToEdit.dueDate;
    }
}
```

This method tried to use the `itemToEdit` instance variable but that ivar does not exist if you depend on automatic property synthesis. When you write `@synthesize itemToEdit;`, the compiler creates a backing ivar with the same name as the property, `itemToEdit`. But the name for the instance variable that automatic synthesize creates is slightly different, it has an additional underscore in front: `_itemToEdit`. Gotcha!

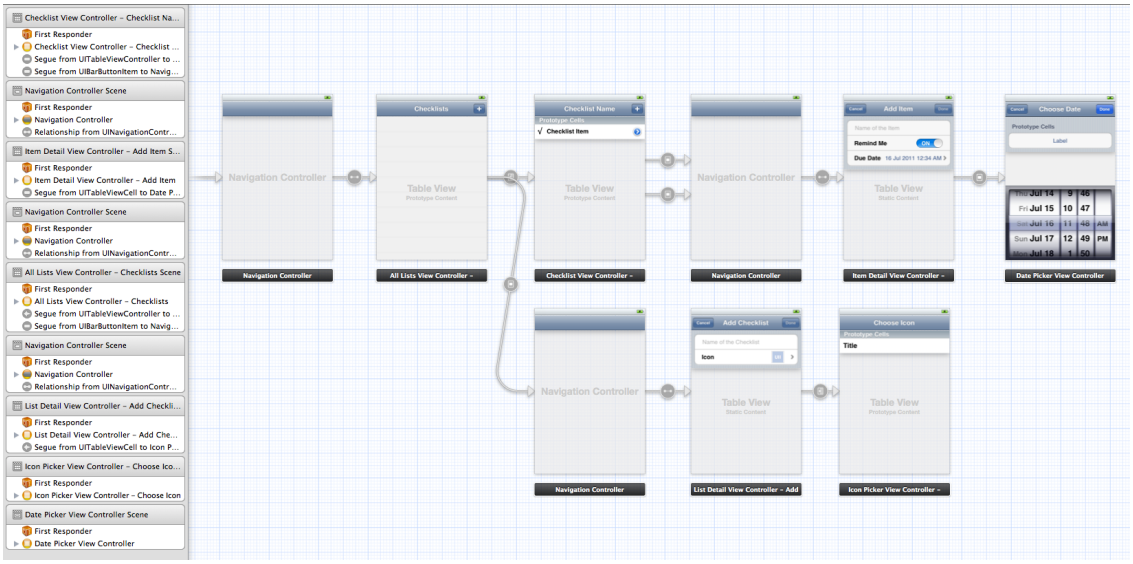
That's a wrap!

Things should be starting to make sense by now. I've sort of thrown you into the deep end by writing an entire app from scratch, and we've touched on a number of advanced topics already, but hopefully you were able to follow along quite well with what I'm doing. If not, then sleep on it for a bit and keep tinkering with the code. Programming requires its own way of thinking and you won't learn that overnight.

This lesson focused mainly on UIKit and its most important controls and patterns. In the next lesson [http://www.raywenderlich.com/ios-apprentice] we'll take a few steps back to talk more about the Objective-C language itself and of course we'll build another cool app.

Here is the final storyboard for Checklists:

The final storyboard



I had trouble fitting that on my screen!

You can find the full source code of the app in the "Source Code" folder for this tutorial.

For more info about table views, see the Table View Programming Guide for iOS [http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/TableView_iPhone/AboutTableViewsiPhone/AboutTableViewsiPhone.html].

For more info about local notifications, see the Local and Push Notification Programming Guide [http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Introduction/Introduction.html].

Icons by The Noun Project [http://thenounproject.com/].

About the author

Matthijs Hollemans is an independent iPad and iPhone developer and designer from the Netherlands. He writes about the technical and non-technical aspects of developing iOS apps on his blog, www.hollance.com [http://www.hollance.com]. He also writes tutorials for raywenderlich.com [http://www.raywenderlich.com].

Feel free to send Matthijs an email [mailto:mail@hollance.com] if you have any questions or comments about these tutorials. And of course you're welcome to visit the forums [http://www.raywenderlich.com/forums/viewforum.php?f=9] for some good conversation.

Thanks for reading!

Revision history

- v1.4 (14 Aug 2012) - Updated for iOS 6. Removed explanation of `viewDidUnload`, as its use is no longer recommended by Apple.
- v1.3 (6 June 2012) - Updated for Xcode 4.3. Added PDF version.
- v1.2 (18 Dec 2011) - Added tutorial 4, StoreSearch
- v1.1 (18 Sept 2011) - Updated for iOS 5
- v1.0 (5 July 2011) - First version (iOS 4)